MATLAB® 7
Programming Fundamentals

MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

MATLAB Programming Fundamentals

Revision History

June 2004	First printing	New for MATLAB 7.0 (Release 14)
October 2004	Online only	Revised for MATLAB 7.0.1 (Release 14SP1)
March 2005	Online only	Revised for MATLAB 7.0.4 (Release 14SP2)
June 2005	Second printing	Minor revision for MATLAB 7.0.4
September 2005	Online only	Revised for MATLAB 7.1 (Release 14SP3)
March 2006	Online only	Revised for MATLAB 7.2 (Release 2006a)
September 2006	Online only	Revised for MATLAB 7.3 (Release 2006b)
March 2007	Online only	Revised for MATLAB 7.4 (Release 2007a)
September 2007	Online only	Revised for Version 7.5 (Release 2007b)
March 2008	Online only	Revised for Version 7.6 (Release 2008a)
October 2008	Online only	Revised for Version 7.7 (Release 2008b)
March 2009	Online only	Revised for Version 7.8 (Release 2009a)

Contents

1 Classes (Data Types)

2 Basic Program Components

3 Functions and Scripts

4 Types of Functions

5 Using Objects

6 Data Import and Export

7 Scientific Data File Formats

8 Error Handling

9 Program Scheduling

10 Performance

11 Memory Usage

12 Programming Tips

Index

Classes (Data Types)

Overview of MATLAB Classes

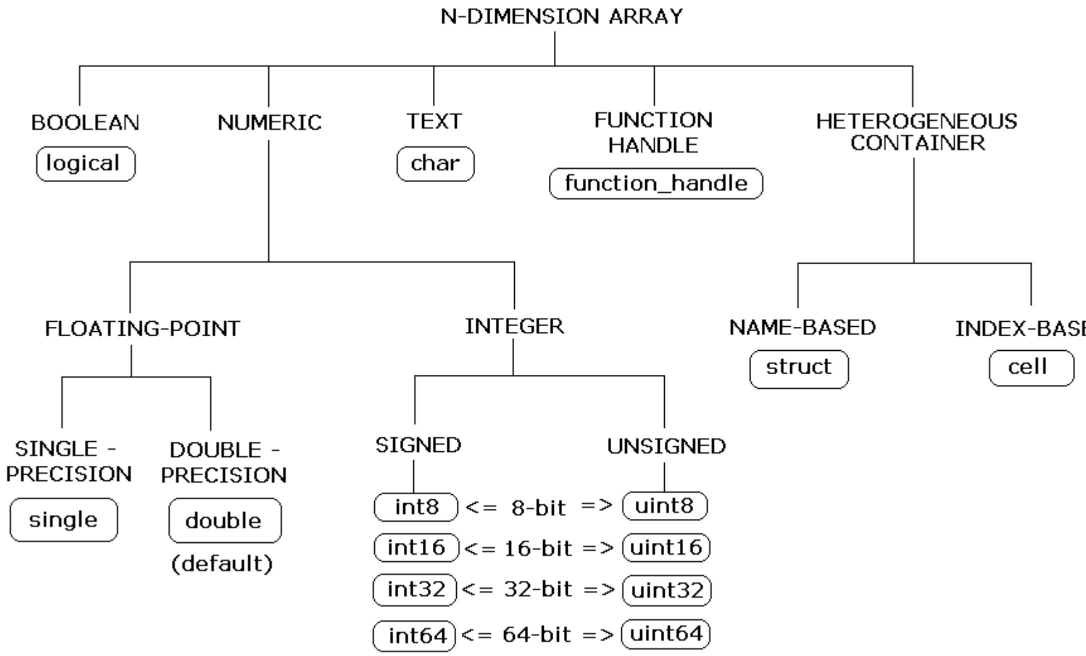
In this section...

Fundamental MATLAB Classes

There are many different data types, or *classes*, that you can work with in the MATLAB® software. You can build matrices and arrays of floating-point and integer data, characters and strings, and logical `true` and `false` states. Function handles connect your code with any MATLAB function regardless of the current scope. Structures and cell arrays, provide a way to store dissimilar types of data in the same array.

There are 15 fundamental classes in MATLAB. Each of these classes is in the form of a matrix or array. This matrix or array is a minimum of 0-by-0 in size and can grow to an n-dimensional array of any size.

All of the fundamental MATLAB classes are circled in the diagram below:



Numeric classes in the MATLAB software include signed and unsigned integers, and single- and double-precision floating-point numbers. By default, MATLAB stores all numeric values as double-precision floating point. (You cannot change the default type and precision.) You can choose to store any number, or array of numbers, as integers or as single-precision. Integer and single-precision arrays offer more memory-efficient storage than double-precision.

All numeric types support basic array operations, such as subscripting and reshaping. All numeric types except for `int64` and `uint64` can be used in mathematical operations.

You can create two-dimensional `double` and `logical` matrices using one of two storage formats: `full` or `sparse`. For matrices with mostly zero-valued elements, a sparse matrix requires a fraction of the storage space required

for an equivalent full matrix. Sparse matrices invoke methods especially tailored to solve sparse problems

These classes require different amounts of storage, the smallest being a `logical` value or 8–bit integer which requires only 1 byte. It is important to keep this minimum size in mind if you work on data in files that were written using a precision smaller than 8 bits.

How to Use the Different Classes

The following table describes these classes in more detail.

Class Name	Documentation	Intended Use
`double`, `single`	Floating-Point Numbers	• Required for fractional numeric data. • Double and Single precision. • Range = 2.2251e-308 to 1.7977e+308. • Two-dimensional arrays can be sparse. • Default numeric type in MATLAB.
`int8, uint8, int16, uint16, int32, uint32, int64, uint64`	Integers	• Use for signed and unsigned whole numbers. • More efficient use of memory. • Range = -263 to 263-1 (64 bit), -231 to 231-1 (32 bit) • Choose from 4 sizes (8, 16, 32, and 64 bits). • Use all but 64-bit in mathematics operations.
`char`	"Characters and Strings" on page 1-39	• Required for text. • Native or unicode. • Converts to/from numeric. • Use with regular expressions. • For multiple strings, use cell arrays.

Class Name	Documentation	Intended Use
`logical`	Logical Classes	• Use in relational conditions or to test state. • Can have one of two values: `true` or `false`. • Also useful in array indexing. • Two-dimensional arrays can be sparse.
function_handle	"Function Handles" on page 1-126	• Pointer to a function. • Enables passing a function to another function • Can also call functions outside usual scope. • Useful in Handle Graphics callbacks. • Save to MAT-file and restore later.
`struct`	Structures	• Fields store arrays of varying classes and sizes. • Access multiple fields/indices in single operation. • Field names identify contents. • Simple method of passing function arguments. • Use in comma-separated lists for efficiency. • More memory required for overhead
`cell`	Cell Arrays	• Cells store arrays of varying classes and sizes. • Allows freedom to package data as you want. • Manipulation of elements is similar to arrays. • Simple method of passing function arguments. • Use in comma-separated lists for efficiency. • More memory required for overhead

Numeric Classes

In this section...

Overview

Numeric classes in the MATLAB software include signed and unsigned integers, and single- and double-precision floating-point numbers. By default, MATLAB stores all numeric values as double-precision floating point. (You cannot change the default type and precision.) You can choose to store any number, or array of numbers, as integers or as single-precision. Integer and single-precision arrays offer more memory-efficient storage than double-precision.

All numeric types support basic array operations, such as subscripting and reshaping. All numeric types except for `int64` and `uint64` can be used in mathematical operations.

Integers

MATLAB has four signed and four unsigned integer classes. Signed types enable you to work with negative integers as well as positive, but cannot represent as wide a range of numbers as the unsigned types because one bit is used to designate a positive or negative sign for the number. Unsigned types give you a wider range of numbers, but these numbers can only be zero or positive.

This section covers:

- "Creating Integer Data" on page 1-7
- "Arithmetic Operations on Integer Classes" on page 1-9
- "Largest and Smallest Values for Integer Classes" on page 1-9
- "Warnings for Integer Classes" on page 1-10
- "Integer Functions" on page 1-13

MATLAB supports 1-, 2-, 4-, and 8-byte storage for integer data. You can save memory and execution time for your programs if you use the smallest integer type that accommodates your data. For example, you don't need a 32-bit integer to store the value `100`.

Here are the eight integer classes, the range of values you can store with each type, and the MATLAB conversion function required to create that type:

Class	Range of Values	Conversion Function
Signed 8-bit integer	-2^7 to 2^7-1	`int8`
Signed 16-bit integer	-2^{15} to 2^{15}-1	`int16`
Signed 32-bit integer	-2^{31} to 2^{31}-1	`int32`
Signed 64-bit integer	-2^{63} to 2^{63}-1	`int64`
Unsigned 8-bit integer	0 to 2^8-1	`uint8`
Unsigned 16-bit integer	0 to 2^{16}-1	`uint16`
Unsigned 32-bit integer	0 to 2^{32}-1	`uint32`
Unsigned 64-bit integer	0 to 2^{64}-1	`uint64`

Creating Integer Data

MATLAB stores numeric data as double-precision floating point (`double`) by default. To store data as an integer, you need to convert from `double` to the desired integer type. Use one of the conversion functions shown in the table above.

For example, to store `325` as a 16-bit signed integer assigned to variable `x`, type

```
x = int16(325);
```

If the number being converted to an integer has a fractional part, MATLAB rounds to the nearest integer. If the fractional part is exactly `0.5`, then from the two equally nearby integers, MATLAB chooses the one for which the absolute value is larger in magnitude:

```
x = 325.499;                  x = x + .001;

int16(x)                      int16(x)
ans =                         ans =
     325                           326
```

If you need to round a number using a rounding scheme other than the default, MATLAB provides four rounding functions: `round`, `fix`, `floor`, and `ceil`. In this example, the `fix` function enables you to override the default and round *towards zero* when the fractional part of a number is `.5`:

```
x = 325.5;

int16(fix(x))
ans =
     325
```

Arithmetic operations that involve both integers and floating-point always result in an integer data type. MATLAB rounds the result, when necessary, according to the default rounding algorithm. The example below yields an exact answer of `1426.75` which MATLAB then rounds to the next highest integer:

```
int16(325) * 4.39
ans =
     1427
```

The integer conversion functions are also useful when converting other classes, such as strings, to integers:

```
str = 'Hello World';

int8(str)
ans =
```

```
72   101   108   108   111    32    87   111   114   108   100
```

Arithmetic Operations on Integer Classes

MATLAB can perform integer arithmetic on the following types of data:

- Integers or integer arrays of the same integer data type. This yields a result that has the same data type as the operands:

  ```
  x = uint32([132 347 528]) .* uint32(75);
  ```

- Integers or integer arrays and scalar double-precision floating-point numbers. This yields a result that has the same data type as the integer operands:

  ```
  x = uint32([132 347 528]) .* 75.49;
  ```

For all binary operations in which one operand is an array of integer data type and the other is a scalar `double`, MATLAB computes the operation using elementwise double-precision arithmetic, and then converts the result back to the original integer data type.

For a list of the operations that support integer classes, see Nondouble Data Type Support in the arithmetic operators reference page.

Largest and Smallest Values for Integer Classes

For each integer data type, there is a largest and smallest number that you can represent with that type. The table shown under "Integers" on page 1-6 lists the largest and smallest values for each integer data type in the "Range of Values" column.

You can also obtain these values with the `intmax` and `intmin` functions:

```
intmax('int8')                 intmin('int8')
ans =                          ans =
   127                            -128
```

If you convert a number that is larger than the maximum value of an integer data type to that type, MATLAB sets it to the maximum value. Similarly, if

you convert a number that is smaller than the minimum value of the integer data type, MATLAB sets it to the minimum value. For example,

```
x = int8(300)                      x = int8(-300)
x =                                x =
   127                                -128
```

Also, when the result of an arithmetic operation involving integers exceeds the maximum (or minimum) value of the data type, MATLAB sets it to the maximum (or minimum) value:

```
x = int8(100) * 3                  x = int8(-100) * 3
x =                                x =
   127                                -128
```

You can make MATLAB return a warning when your input is outside the range an integer data type. This is described in the next section.

Warnings for Integer Classes

Use the `intwarning` function to make MATLAB return a warning message when it converts a number outside the range of an integer data type to that type, or when the result of an arithmetic operation overflows. There are four possible warning messages that you can turn on using `intwarning`:

Message Identifier	Reason for Warning
`MATLAB:intConvertOverflow`	Overflow when attempting to convert from a numeric class to an integer class
`MATLAB:intMathOverflow`	Overflow when attempting an integer arithmetic operation
`MATLAB:intConvertNonIntVal`	Attempt to convert a noninteger value to an integer
`MATLAB:intConvertNaN`	Attempt to convert NaN (Not a Number) to an integer

Querying the Present Warning State. Use the following command to display the state of all integer warnings:

```
intwarning('query')
   The state of warning 'MATLAB:intConvertNaN' is 'off'.
   The state of warning 'MATLAB:intConvertNonIntVal' is 'off'.
   The state of warning 'MATLAB:intConvertOverflow' is 'off'.
   The state of warning 'MATLAB:intMathOverflow' is 'off'.
```

To display the state of one or more selected warnings, index into the structure returned by `intwarning`. This example shows the current state of the `intConvertOverflow` warning:

```
iwState = intwarning('query');
iwState(3)
ans =
    identifier: 'MATLAB:intConvertOverflow'
         state: 'off'
```

Turning the Warning On. To enable all four integer warnings, use `intwarning` with the string `'on'`:

```
intwarning('on');
intwarning('query')
   The state of warning 'MATLAB:intConvertNaN' is 'on'.
   The state of warning 'MATLAB:intConvertNonIntVal' is 'on'.
   The state of warning 'MATLAB:intConvertOverflow' is 'on'.
   The state of warning 'MATLAB:intMathOverflow' is 'on'.
```

To enable one or more selected integer warnings, first make sure that all integer warnings are disabled:

```
intwarning('off');
```

Note that, in this state, the following conversion to a 16-bit integer overflows, but does not issue a warning:

```
x = int16(50000)
x =
  32767
```

Find which of the four warnings covers integer conversion. In this case, it is the third in the structure array:

```
iwState = intwarning('query');
iwState(3).identifier
ans =
   MATLAB:intConvertOverflow
```

Set the warning state to `'on'` in the structure, and then call `intwarning` using the structure as input:

```
iwState(3).state = 'on';
intwarning(iwState);
```

With the warning enabled, the overflow on conversion does issue the warning message:

```
x = int16(50000)
Warning: Out of range value converted to intmin('int16') or
intmax('int16').
x =
  32767
```

You can also use the following `for` loop to enable integer warnings selectively:

```
maxintwarn = 4;

for k = 1:maxintwarn
   if strcmp(iwState(k).identifier, 'MATLAB:intConvertOverflow')
      iwState(k).state = 'on';
      intwarning(iwState);
   end
end
```

Turning the Warning Off. To turn all integer warnings off (their default state when you start MATLAB), enter

```
intwarning('off')
```

To disable selected integer warnings, follow the steps shown for enabling warnings, but with the `state` field of the structure set to `'off'`:

```
iwState(3).identifier
ans =
```

```
    MATLAB:intConvertOverflow

iwState(3).state = 'off';
intwarning(iwState);
```

Turning Warnings On or Off Temporarily. When writing M-files that contain integer classes, it is sometimes convenient to temporarily turn integer warnings on, and then return the states of the warnings (`'on'` or `'off'`) to their previous settings. The following commands illustrate how to do this:

```
oldState = intwarning('on');

int8(200);
Warning: Out of range value converted to intmin('int8') or
intmax('int8').

intwarning(oldState)
```

To temporarily turn the warnings off, change the first line of the preceding code to

```
oldState = intwarning('off');
```

Recommended Usage of Math Overflow Warning. Enabling the `MATLAB:intMathOverflow` warning slows down integer arithmetic. It is recommended that you enable this particular warning only when you need to diagnose unusual behavior in your code, and disable it during normal program operation. The other integer warnings listed above do not affect program performance.

Note that calling `intwarning('on')` enables all four integer warnings, including the `intMathOverflow` warning that can have an effect on integer arithmetic.

Integer Functions

See Integer Functions on page 1-30 for a list of functions most commonly used with integers in MATLAB.

Floating-Point Numbers

MATLAB represents floating-point numbers in either double-precision or single-precision format. The default is double precision, but you can make any number single precision with a simple conversion function.

This section covers:

Double-Precision Floating Point

MATLAB constructs the double-precision (or `double`) data type according to IEEE® Standard 754 for double precision. Any value stored as a `double` requires 64 bits, formatted as shown in the table below:

Bits	Usage
63	Sign (`0` = positive, `1` = negative)
62 to 52	Exponent, biased by `1023`
51 to 0	Fraction `f` of the number `1.f`

Single-Precision Floating Point

MATLAB constructs the single-precision (or `single`) data type according to IEEE Standard 754 for single precision. Any value stored as a `single` requires 32 bits, formatted as shown in the table below:

Bits	Usage
31	Sign (0 = positive, 1 = negative)
30 to 23	Exponent, biased by 127
22 to 0	Fraction `f` of the number `1.f`

Because MATLAB stores numbers of type `single` using 32 bits, they require less memory than numbers of type `double`, which use 64 bits. However, because they are stored with fewer bits, numbers of type `single` are represented to less precision than numbers of type `double`.

Creating Floating-Point Data

Use double-precision to store values greater than approximately 3.4 x 10^{38} or less than approximately -3.4 x 10^{38}. For numbers that lie between these two limits, you can use either double- or single-precision, but single requires less memory.

Creating Double-Precision Data. Because the default numeric type for MATLAB is `double`, you can create a `double` with a simple assignment statement:

```
x = 25.783;
```

The `whos` function shows that MATLAB has created a 1-by-1 array of type `double` for the value you just stored in `x`:

```
whos x
  Name      Size                    Bytes  Class

  x         1x1                         8  double
```

Use `isfloat` if you just want to verify that `x` is a floating-point number. This function returns logical 1 (`true`) if the input is a floating-point number, and logical 0 (`false`) otherwise:

```
isfloat(x)
ans =
     1
```

You can convert other numeric data, characters or strings, and logical data to double precision using the MATLAB function, `double`. This example converts a signed integer to double-precision floating point:

```
y = int64(-589324077574);          % Create a 64-bit integer

x = double(y)                      % Convert to double
x =
   -5.8932e+011
```

Creating Single-Precision Data. Because MATLAB stores numeric data as a `double` by default, you need to use the `single` conversion function to create a single-precision number:

```
x = single(25.783);
```

The `whos` function returns the attributes of variable `x` in a structure. The `bytes` field of this structure shows that when `x` is stored as a single, it requires just 4 bytes compared with the 8 bytes to store it as a `double`:

```
xAttrib = whos('x');
xAttrib.bytes
ans =
     4
```

You can convert other numeric data, characters or strings, and logical data to single precision using the `single` function. This example converts a signed integer to single-precision floating point:

```
y = int64(-589324077574);          % Create a 64-bit integer

x = single(y)                      % Convert to single
x =
   -5.8932e+011
```

Arithmetic Operations on Floating-Point Numbers

This section describes which classes you can use in arithmetic operations with floating-point numbers.

Double-Precision Operations. You can perform basic arithmetic operations with `double` and any of the following other classes. When one or more operands is an integer (scalar or array), the `double` operand must be a scalar. The result is of type `double`, except where noted otherwise:

- `single` — The result is of type `single`
- `double`
- `int*` or `uint*` — The result has the same data type as the integer operand
- `char`
- `logical`

This example performs arithmetic on data of types `char` and `double`. The result is of type `double`:

```
c = 'uppercase' - 32;

class(c)
ans =
   double

char(c)
ans =
   UPPERCASE
```

Single-Precision Operations. You can perform basic arithmetic operations with `single` and any of the following other classes. The result is always `single`:

- `single`
- `double`
- `char`
- `logical`

In this example, 7.5 defaults to type `double`, and the result is of type `single`:

```
x = single([1.32 3.47 5.28]) .* 7.5;

class(x)
ans =
   single
```

Largest and Smallest Values for Floating-Point Classes

For the `double` and `single` classes, there is a largest and smallest number that you can represent with that type.

Largest and Smallest Double-Precision Values. The MATLAB functions `realmax` and `realmin` return the maximum and minimum values that you can represent with the `double` data type:

```
str = 'The range for double is:\n\t%g to %g and\n\t %g to  %g';
sprintf(str, -realmax, -realmin, realmin, realmax)

ans =
The range for double is:
   -1.79769e+308 to -2.22507e-308 and
    2.22507e-308 to  1.79769e+308
```

Numbers larger than `realmax` or smaller than `-realmax` are assigned the values of positive and negative infinity, respectively:

```
realmax + .0001e+308
ans =
   Inf

-realmax - .0001e+308
ans =
  -Inf
```

Largest and Smallest Single-Precision Values. The MATLAB functions `realmax` and `realmin`, when called with the argument `'single'`, return the maximum and minimum values that you can represent with the `single` data type:

```
str = 'The range for single is:\n\t%g to %g and\n\t %g to  %g';
sprintf(str, -realmax('single'), -realmin('single'), ...
    realmin('single'), realmax('single'))

ans =
The range for single is:
   -3.40282e+038 to -1.17549e-038 and
    1.17549e-038 to  3.40282e+038
```

Numbers larger than `realmax`('single') or smaller than `-realmax` ('single') are assigned the values of positive and negative infinity, respectively:

```
realmax('single') + .0001e+038
ans =
   Inf

-realmax('single') - .0001e+038
ans =
  -Inf
```

Accuracyof Floating-Point Data

If the result of a floating-point arithmetic computation is not as precise as you had expected, it is likely caused by the limitations of your computer's hardware. Probably, your result was a little less exact because the hardware had insufficient bits to represent the result with perfect accuracy; therefore, it truncated the resulting value.

Double-Precision Accuracy. Because there are only a finite number of double-precision numbers, you cannot represent all numbers in double-precision storage. On any computer, there is a small gap between each double-precision number and the next larger double-precision number. You can determine the size of this gap, which limits the precision of your results, using the `eps` function. For example, to find the distance between `5` and the next larger double-precision number, enter

```
format long

eps(5)
ans =
```

```
    8.881784197001252e-016
```

This tells you that there are no double-precision numbers between 5 and `5 + eps(5)`. If a double-precision computation returns the answer 5, the result is only accurate to within `eps(5)`.

The value of `eps(x)` depends on `x`. This example shows that, as `x` gets larger, so does `eps(x)`:

```
eps(50)
ans =
    7.105427357601002e-015
```

If you enter `eps` with no input argument, MATLAB returns the value of `eps(1)`, the distance from `1` to the next larger double-precision number.

Single-Precision Accuracy. Similarly, there are gaps between any two single-precision numbers. If `x` has type `single`, `eps(x)` returns the distance between `x` and the next larger single-precision number. For example,

```
x = single(5);
eps(x)
```

returns

```
ans =
  4.7684e-007
```

Note that this result is larger than `eps(5)`. Because there are fewer single-precision numbers than double-precision numbers, the gaps between the single-precision numbers are larger than the gaps between double-precision numbers. This means that results in single-precision arithmetic are less precise than in double-precision arithmetic.

For a number `x` of type `double`, `eps(single(x))` gives you an upper bound for the amount that `x` is rounded when you convert it from `double` to `single`. For example, when you convert the double-precision number `3.14` to `single`, it is rounded by

```
double(single(3.14) - 3.14)
ans =
```

```
  1.0490e-007
```

The amount that 3.14 is rounded is less than

```
eps(single(3.14))
ans =
  2.3842e-007
```

Avoiding Common Problems with Floating-Point Arithmetic

Almost all operations in MATLAB are performed in double-precision arithmetic conforming to the IEEE standard 754. Because computers only represent numbers to a finite precision (double precision calls for 52 mantissa bits), computations sometimes yield mathematically nonintuitive results. It is important to note that these results are not bugs in MATLAB.

Use the following examples to help you identify these cases:

Example 1 — Round-Off or What You Get Is Not What You Expect.

The decimal number 4/3 is not exactly representable as a binary fraction. For this reason, the following calculation does not give zero, but rather reveals the quantity eps.

```
e = 1 - 3*(4/3 - 1)

e =
  2.2204e-016
```

Similarly, 0.1 is not exactly representable as a binary number. Thus, you get the following nonintuitive behavior:

```
a = 0.0;
for i = 1:10
  a = a + 0.1;
end
a == 1

ans =
     0
```

Note that the order of operations can matter in the computation:

```
b = 1e-16 + 1 - 1e-16;
c = 1e-16 - 1e-16 + 1;
b == c

ans =
     0
```

There are gaps between floating-point numbers. As the numbers get larger, so do the gaps, as evidenced by:

```
(2^53 + 1) - 2^53

ans =
     0
```

Since `pi` is not really `pi`, it is not surprising that `sin(pi)` is not exactly zero:

```
sin(pi)

ans =
   1.224646799147353e-016
```

Example 2 — Catastrophic Cancellation. When subtractions are performed with nearly equal operands, sometimes cancellation can occur unexpectedly. The following is an example of a cancellation caused by swamping (loss of precision that makes the addition insignificant).

```
sqrt(1e-16 + 1) - 1

ans =
     0
```

Some functions in MATLAB, such as `expm1` and `log1p`, may be used to compensate for the effects of catastrophic cancellation.

Example 3 — Floating-Point Operations and Linear Algebra. Round-off, cancellation, and other traits of floating-point arithmetic combine to produce startling computations when solving the problems of linear algebra. MATLAB warns that the following matrix `A` is ill-conditioned, and therefore the system `Ax = b` may be sensitive to small perturbations. Although the computation differs from what you expect in exact arithmetic, the result is correct.

```
A = [2 eps -eps; eps 1 1; -eps 1 1];
b = [2; eps + 2; -eps + 2];
x = A\b

x =
  1.0e+015 *
   0.000000000000001
   2.251799813685249
  -2.251799813685247
```

These are only a few of the examples showing how IEEE floating-point arithmetic affects computations in MATLAB. Note that all computations performed in IEEE 754 arithmetic are affected, this includes applications written in C or FORTRAN, as well as MATLAB. For more examples and information, see `Technical Note 1108   Common Problems with Floating-Point Arithmetic`.

Floating-Point Functions

See Floating-Point Functions on page 1-30 for a list of functions most commonly used with floating-point numbers in MATLAB.

References

The following references provide more information about floating-point arithmetic.

[1] Moler, Cleve, “Floating Points,” *MATLAB News and Notes*, Fall, 1996. A PDF version is available on the MathWorks Web site at http://www.mathworks.com/company/newsletters/news_notes/pdf/Fall96Cleve.pdf

[2] Moler, Cleve, *Numerical Computing with MATLAB*, S.I.A.M.
A PDF version is available on the MathWorks Web site at
http://www.mathworks.com/moler/.

Complex Numbers

Complex numbers consist of two separate parts: a real part and an imaginary part. The basic imaginary unit is equal to the square root of `-1`. This is represented in MATLAB by either of two letters: `i` or `j`.

Creating Complex Numbers

The following statement shows one way of creating a complex value in MATLAB. The variable `x` is assigned a complex number with a real part of `2` and an imaginary part of `3`:

```
x = 2 + 3i;
```

Another way to create a complex number is using the `complex` function. This function combines two numeric inputs into a complex output, making the first input real and the second imaginary:

```
x = rand(3) * 5;
y = rand(3) * -8;

z = complex(x, y)
z =
   4.7842 -1.0921i   0.8648 -1.5931i   1.2616 -2.2753i
   2.6130 -0.0941i   4.8987 -2.3898i   4.3787 -3.7538i
   4.4007 -7.1512i   1.3572 -5.2915i   3.6865 -0.5182i
```

You can separate a complex number into its real and imaginary parts using the `real` and `imag` functions:

```
zr = real(z)
zr =
    4.7842    0.8648    1.2616
    2.6130    4.8987    4.3787
    4.4007    1.3572    3.6865

zi = imag(z)
```

```
zi =
   -1.0921   -1.5931   -2.2753
   -0.0941   -2.3898   -3.7538
   -7.1512   -5.2915   -0.5182
```

Complex Number Functions

See Complex Number Functions on page 1-31 for a list of functions most commonly used with MATLAB complex numbers in MATLAB.

Infinity and NaN

MATLAB uses the special values `inf`, `-inf`, and `NaN` to represent values that are positive and negative infinity, and not a number respectively.

Infinity

MATLAB represents infinity by the special value `inf`. Infinity results from operations like division by zero and overflow, which lead to results too large to represent as conventional floating-point values. MATLAB also provides a function called `inf` that returns the IEEE arithmetic representation for positive infinity as a `double` scalar value.

Several examples of statements that return positive or negative infinity in MATLAB are shown here.

`x = 1/0`
`x =`
`Inf`
	`x = 1.e1000`
	`x =`
	`   Inf`

`x = exp(1000)`
`x =`
`   Inf`
	`x = log(0)`
	`x =`
	`   -Inf`

Use the `isinf` function to verify that `x` is positive or negative infinity:

```
x = log(0);

isinf(x)
ans =
```

```
     1
```

NaN

MATLAB represents values that are not real or complex numbers with a special value called `NaN`, which stands for Not a Number. Expressions like `0/0` and `inf/inf` result in `NaN`, as do any arithmetic operations involving a `NaN`.

For example, the statement `n/0`, where `n` is complex, returns `NaN` for the real part of the result:

```
x = 7i/0
x =
      NaN +    Infi
```

Use the `isnan` function to verify that the real part of `x` is NaN:

```
isnan(real(x))
ans =
     1
```

MATLAB also provides a function called `NaN` that returns the IEEE arithmetic representation for `NaN` as a `double` scalar value:

```
x = NaN;

whos x
  Name      Size                    Bytes  Class

  x         1x1                         8  double
```

Logical Operations on NaN. Because two `NaN`s are not equal to each other, logical operations involving `NaN` always return false, except for a test for inequality, (`NaN ~= NaN`):

```
NaN > NaN
ans =
     0

NaN ~= NaN
```

```
ans =
     1
```

Infinity and NaN Functions

See Infinity and NaN Functions on page 1-31 for a list of functions most commonly used with `inf` and `NaN` in MATLAB.

Identifying Numeric Classes

You can check the data type of a variable `x` using any of these commands.

Command	Operation
`whos x`	Display the data type of `x`.
`xType = class(x);`	Assign the data type of `x` to a variable.
`isnumeric(x)`	Determine if `x` is a numeric type.

`isa(x, 'integer')`
`isa(x, 'uint64')`
`isa(x, 'float')`
`isa(x, 'double')`
`isa(x, 'single')`
	Determine if `x` is the specified numeric type. (Examples for any integer, unsigned 64-bit integer, any floating point, double precision, and single precision are shown here).

`isreal(x)`	Determine if `x` is real or complex.
`isnan(x)`	Determine if `x` is Not a Number (`NaN`).
`isinf(x)`	Determine if `x` is infinite.
`isfinite(x)`	Determine if `x` is finite.

Display Format for Numeric Values

By default, MATLAB displays numeric output as 5-digit scaled, fixed-point values. You can change the way numeric values are displayed to any of the following:

- 5-digit scaled fixed point, floating point, or the best of the two
- 15-digit scaled fixed point, floating point, or the best of the two

- A ratio of small integers
- Hexadecimal (base 16)
- Bank notation

All available formats are listed on the `format` reference page.

To change the numeric display setting, use either the `format` function or the **Preferences** dialog box (accessible from the MATLAB **File** menu). The `format` function changes the display of numeric values for the duration of a single MATLAB session, while your Preferences settings remain active from one session to the next. These settings affect only how numbers are displayed, not how MATLAB computes or saves them.

Display Format Examples

Here are a few examples of the various formats and the output produced from the following two-element vector `x`, with components of different magnitudes.

Check the current format setting:

```
get(0, 'format')
ans =
   short
```

Set the value for `x` and display in 5-digit scaled fixed point:

```
x = [4/3 1.2345e-6]
x =
    1.3333    0.0000
```

Set the format to 5-digit floating point:

```
format short e
x
x =
  1.3333e+000  1.2345e-006
```

Set the format to 15-digit scaled fixed point:

```
format long
```

```
x
x =
   1.33333333333333   0.00000123450000
```

Set the format to `'rational'` for small integer ratio output:

```
format rational
x
x =
     4/3          1/810045
```

Set an integer value for `x` and display it in hexadecimal (base 16) format:

```
format hex
x = uint32(876543210)
x =
   343efcea
```

Setting Numeric Format in a Program

To temporarily change the numeric format inside a program, get the original format using the `get` function and save it in a variable. When you finish working with the new format, you can restore the original format setting using the `set` function as shown here:

```
origFormat = get(0, 'format');
format('rational');

  -- Work in rational format --

set(0,'format', origFormat);
```

Function Summary

MATLAB provides these functions for working with numeric classes:

- Integer Functions on page 1-30
- Floating-Point Functions on page 1-30
- Complex Number Functions on page 1-31

- Infinity and NaN Functions on page 1-31
- Class Identification Functions on page 1-32
- Output Formatting Functions on page 1-32

Integer Functions

Function	Description
`int8, int16, int32, int64`	Convert to signed 1-, 2-, 4-, or 8-byte integer.
`uint8, uint16, uint32, uint64`	Convert to unsigned 1-, 2-, 4-, or 8-byte integer.
`ceil`	Round towards plus infinity to nearest integer
`class`	Return the data type of an object.
`fix`	Round towards zero to nearest integer
`floor`	Round towards minus infinity to nearest integer
`isa`	Determine if input value has the specified data type.
`isinteger`	Determine if input value is an integer array.
`isnumeric`	Determine if input value is a numeric array.
`round`	Round towards the nearest integer

Floating-Point Functions

Function	Description
`double`	Convert to double precision.
`single`	Convert to single precision.
`class`	Return the data type of an object.
`isa`	Determine if input value has the specified data type.
`isfloat`	Determine if input value is a floating-point array.
`isnumeric`	Determine if input value is a numeric array.

Floating-Point Functions (Continued)

Function	Description
`eps`	Return the floating-point relative accuracy. This value is the tolerance MATLAB uses in its calculations.
`realmax`	Return the largest floating-point number your computer can represent.
`realmin`	Return the smallest floating-point number your computer can represent.

Complex Number Functions

Function	Description
`complex`	Construct complex data from real and imaginary components.
`i` or `j`	Return the imaginary unit used in constructing complex data.
`real`	Return the real part of a complex number.
`imag`	Return the imaginary part of a complex number.
`isreal`	Determine if a number is real or imaginary.

Infinity and NaN Functions

Function	Description
`inf`	Return the IEEE value for infinity.
`isnan`	Detect `NaN` elements of an array.
`isinf`	Detect infinite elements of an array.
`isfinite`	Detect finite elements of an array.
`nan`	Return the IEEE value for Not a Number.

Class Identification Functions

Function	Description
`class`	Return data type (or class).
`isa`	Determine if input value is of the specified data type.
`isfloat`	Determine if input value is a floating-point array.
`isinteger`	Determine if input value is an integer array.
`isnumeric`	Determine if input value is a numeric array.
`isreal`	Determine if input value is real.
`whos`	Display the data type of input.

Output Formatting Functions

Function	Description
`format`	Control display format for output.

Logical Classes

In this section...

Overview of Logical Classes

The `logical` data type represents a logical `true` or `false` state using the numbers `1` and `0`, respectively. Certain MATLAB functions and operators return logical `true` or `false` to indicate whether a certain condition was found to be true or not. For example, the statement `50>40` returns a logical `true` value.

Logical data does not have to be scalar; MATLAB supports arrays of logical values as well. For example, the following statement returns a vector of logicals indicating `false` for the first two elements and `true` for the last three:

```
[30 40 50 60 70] > 40
ans =
     0     0     1     1     1
```

This statement returns a 4-by-4 array of logical values:

```
x = magic(4) >= 9
x =
     1     0     0     1
     0     1     1     0
     1     0     0     1
     0     1     1     0
```

The MATLAB functions that have names beginning with `is` (e.g., `ischar`, `issparse`) also return a logical value or array:

```
a = [2.5 6.7 9.2 inf 4.8];
```

```
isfinite(a)
ans =
     1     1     1     0     1
```

Logical arrays can also be sparse as long as they have no more than two dimensions:

```
x = sparse(magic(20) > 395)
x =
   (1,1)        1
   (1,4)        1
   (1,5)        1
  (20,18)       1
  (20,19)       1
```

Identifying Logical Arrays

This table shows the commands you can use to determine whether or not an array `x` is logical. The last function listed, `cellfun`, operates on cell arrays, which you can read about in the section on cell arrays.

Command	Operation
`whos(x)`	Display value and data type for `x`.
`islogical(x)`	Return `true` if array is logical.
`isa(x, 'logical')`	Return `true` if array is logical.
`class(x)`	Return string with data type name.
`cellfun('islogical', x)`	Check each cell array element for logical.

Examples of Identifying Logical Arrays

Create a 3-by-6 array of logicals and use the `whos` function to identify the size, byte count, and class (i.e., data type) of the array.

```
% Initialize the state of the random number generator.
rand('state',0);
A = rand(3,6) > .5
A =
```

```
     1     0     0     0     1     0
     0     1     0     1     1     1
     1     1     1     1     0     1

whos A
  Name      Size            Bytes  Class      Attributes

  A         3x6                18  logical
```

Find the class of each of these expressions:

```
B = logical(-2.8);  C = false;  D = 50>40;  E = isinteger(4.9);

whos B C D E
  Name      Size            Bytes  Class      Attributes

  B         1x1                 1  logical
  C         1x1                 1  logical
  D         1x1                 1  logical
  E         1x1                 1  logical
```

Display the `class` of `A`:

```
% Initialize the state of the random number generator.
rand('state',0);
A = rand(3,6) > .5

fprintf('A is a %s\n', class(A))
   A is a logical
```

Create cell array C and use `islogical` to identify the logical elements:

```
C = {1, 0, true, false, pi, A};
cellfun('islogical', C)
ans =
     0     0     1     1     0     1
```

Functions that Return a Logical Result

This table shows some of the MATLAB operations that return a logical `true` or `false`. Most mathematics operations are not supported on logical values.

Function	Operation
true, false	Setting value to true or false
logical	Numeric to logical conversion
& (and), | (or), ~ (not), xor, any, all	Logical operations
&&, ||	Short-circuit AND and OR
== (eq), ~= (ne), < (lt), > (gt), <= (le), >= (ge)	Relational operations
All is* functions, cellfun	Test operations
strcmp, strncmp, strcmpi, strncmpi	String comparisons

Examples of Functions that Return a Logical Result

MATLAB functions that test the state of a variable or expression return a logical result:

```
A = isstrprop('abc123def', 'alpha')
A =
     1     1     1     0     0     0     1     1      1
```

Logical functions such as xor return a logical result:

```
xor([1 0 'ab' 2.4], [ 0 0 'ab', 0])
ans =
     1     0     0     1
```

Note however that the bitwise operators do not return a logical:

```
X = bitxor(3, 12);
whos X
  Name      Size            Bytes  Class     Attributes

  X         1x1                 8  double
```

String comparison functions also return a logical:

```
S = 'D:\matlab\mfiles\test19.m';
strncmp(S, 'D:\matlab', 9)
ans =
```

```
     1
```

Note the difference between the elementwise and short-circuit logical operators. Short-circuit operators, such as `&&` and `||`, test only as much of the input expression as necessary. In the second part of this example, it makes no difference that `B` is undefined because the state of `A` alone determines that the expression is `false`:

```
A = 0;
A & B
??? Undefined function or variable 'B'.

A && B
ans =
     0
```

One way of implementing an infinite loop is to use the `while` function along with the logical constant `true`:

```
while true
a = [];  b = [];
a = input('Enter username: ', 's');

   if ~isempty(a)
   b = input('Enter password: ', 's');
   end

   if ~isempty(b)
   disp 'Attempting to log in to account ...'
   break
   end
end
```

Using Logical Arrays in Conditional Statements

Conditional statements are useful when you want to execute a block of code only when a certain condition is met. For example, the `sprintf` command shown below is valid only if `str` is a nonempty string:

```
str = input('Enter input string: ', 's');
if ~isempty(str) && ischar(str)
   sprintf('Input string is ''%s''', str)
```

```
end
```

Now run the code:

```
Enter input string: Hello
ans =
   Input string is 'Hello'
```

Using Logical Arrays in Indexing

A logical matrix provides a different type of array indexing in MATLAB. While most indices are numeric, indicating a certain row or column number, logical indices are positional. That is, it is the *position* of each 1 in the logical matrix that determines which array element is being referred to.

See "Using Logicals in Array Indexing" for more information on this subject.

Characters and Strings

In this section...

Creating Character Arrays

A character in the MATLAB software is actually an integer value converted to its Unicode® character equivalent. A character string is a vector with components that are the numeric codes for the characters. The actual characters displayed depend on the character set encoding for a given font.

The elements of a character or string belong to the `char` class. Arrays of class `char` can hold multiple strings, as long as each string in the array has the same length. (This is because MATLAB arrays must be rectangular.) To store an array of strings of unequal length, use a cell array.

Creating a Single Character

Store a single character in the MATLAB workspace by enclosing the character in single quotation marks and assigning it to a variable:

```
hChar = 'h';
```

This creates a 1-by-1 matrix of class `char`. Each character occupies 2 bytes of workspace memory:

```
whos hChar
```

```
  Name        Size             Bytes  Class    Attributes

  hChar       1x1                  2  char
```

The numeric value of hChar is 104:

```
uint8(hChar)
ans =
   104
```

Creating a Character String

Create a string by enclosing a sequence of letters in single quotation marks. MATLAB represents the five-character string shown below as a 1-by-5 vector of class char. It occupies 2 bytes of memory for each character in the string:

```
str = 'Hello';

whos str
  Name       Size             Bytes  Class    Attributes

  str        1x5                 10  char
```

The uint8 function converts characters to their numeric values:

```
str_numeric = uint8(str)
str_numeric =
   72  101  108  108  111
```

The char function converts the integer vector back to characters:

```
str_alpha = char([72 101 108 108 111])
str_alpha =
   Hello
```

Creating an Array of Strings

Create an array of strings in the same way that you would create a numeric array. Use the array constructor ([ ]), delimit each row in the array with a semicolon, and enclose each string in single quotation marks. Like numeric

arrays, character arrays must be rectangular. That is, each row of the array must be the same length:

```
name = ['Thomas R. Lee'; ...
        'Sr. Developer'; ...
        'SFTware Corp.'];
```

Padding Strings. To make an array from strings that are originally of unequal length, you must either pad the shorter strings with space characters, or use a cell array. If you choose to pad the strings, there are two ways to do this. You can either add space characters manually, as shown here:

```
name = ['Harold A. Jorgensen      '; ...
        'Assistant Project Manager'; ...
        'SFTware Corp.            '];
```

or construct the array using the `char` function. This function automatically pads the shorter strings with spaces at the end. This array now consists of three strings of 25 characters each:

```
name = char('Harold A. Jorgensen', ...
   'Assistant Project Manager', 'SFTware Corp.');

size(name)
ans =
     3    25
```

Creating Character Arrays by Concatenation

You can join two or more character arrays together to create a new character array. This is called *concatenation* and is explained for numeric arrays in the section "Concatenating Matrices". To do this, use either the string concatenation function, `strcat`, or the MATLAB concatenation operator, `[]`. The latter method preserves any trailing spaces found in the input arrays; the former method does not:

```
name =    'Thomas R. Lee';
title =   'Sr. Developer';
company = 'SFTware Corp.';

s = strcat(name, ', ', title, ', ', company);
```

```
s = [name, ', ', title, ', ', company];
```

To concatenate strings vertically, use either the `strvcat` function or the `[]` operator with semicolons separating the rows:

```
s = strvcat(name, title, company);
s = [name; title; company];
```

This command concatenates the value assigned to keyword `matlabroot` with the remainder of a path string:

```
dir([matlabroot '\extern\examples\mex\yprime.c'])
   yprime.c
```

Identifying Characters in a String

Use any of the following functions to identify a character or string, or certain characters in a string:

Function	Description
`ischar`	Determine whether the input is a character array.
`isletter`	Find all alphabetic letters in the input string.
`isspace`	Find all space characters in the input string.
`isstrprop`	Find all characters of a specific category.

```
str = 'Find the space characters in this string';
%          |   |     |          |  |    |
%          5   9    15         26 29   34

find(isspace(str))
ans =
     5     9    15    26    29    34
```

Working with Space Characters

The `blanks` function creates a string of space characters. The following example creates a string of 15 space characters:

```
s = blanks(15)
```

```
s =
```

To make the example more useful, append a '|' character to the beginning and end of the blank string so that you can see the output:

```
['|' s '|']         % Make result visible.
ans =
   |               |
```

Insert a few nonspace characters in the middle of the blank string:

```
s(6:10) = 'AAAAA';

['|' s '|']         % Make result visible.
ans =
   |     AAAAA     |
```

You can justify the positioning of these characters to the left or right using the strjust function:

```
sLeft = strjust(s, 'left');

['|' sLeft '|']         % Make result visible.
ans =
   |AAAAA          |

sRight = strjust(s, 'right');

['|' sRight '|']         % Make result visible.
ans =
   |          AAAAA|
```

Remove all trailing space characters with deblank:

```
sDeblank = deblank(s);

['|' sDeblank '|']         % Make result visible.
ans =
   |     AAAAA|
```

Remove all leading and trailing spaces with `strtrim`:

```
sTrim = strtrim(s);

['|' sTrim '|']         % Make result visible.
ans =
   |AAAAA|
```

Expanding Character Arrays

Generally the MathWorks does not recommend expanding the size of an existing character array by assigning additional characters to indices beyond the bounds of the array such that part of the array becomes padded with zeros.

Cell Arrays of Strings

Creating strings in a regular MATLAB array requires that all strings in the array be of the same length. This often means that you have to pad blanks at the end of strings to equalize their length. However, another type of MATLAB array, the cell array, can hold different sizes and types of data in an array without padding. Cell arrays provide a more flexible way to store strings of varying length.

For details on cell arrays, see Cell Arrays in the Programming Fundamentals documentation.

Converting to a Cell Array of Strings

The `cellstr` function converts a character array into a cell array of strings. Consider this character array:

```
data = ['Allison Jones';'Development  ';'Phoenix      '];
```

Each row of the matrix is padded so that all have equal length (in this case, 13 characters).

Now use `cellstr` to create a column vector of cells, each cell containing one of the strings from the `data` array:

```
celldata = cellstr(data)
celldata =
```

```
    'Allison Jones'
    'Development'
    'Phoenix'
```

Note that the `cellstr` function strips off the blanks that pad the rows of the input string matrix:

```
length(celldata{3})
ans =
    7
```

The `iscellstr` function determines if the input argument is a cell array of strings. It returns a logical `1` (`true`) in the case of `celldata`:

```
iscellstr(celldata)
ans =
     1
```

Use `char` to convert back to a standard padded character array:

```
strings = char(celldata)
strings =
   Allison Jones
   Development
   Phoenix

length(strings(3,:))
ans =
    13
```

Functions for Cell Arrays of Strings

This table describes the MATLAB functions for working with cell arrays.

Function	Description
`cellstr`	Convert a character array to a cell array of strings.
`char`	Convert a cell array of strings to a character array.
`deblank`	Remove trailing blanks from a string.

Function	Description
`iscellstr`	Return `true` for acell array of strings.
`sort`	Sort elements in ascending or descending order.
`strcat`	Concatenate strings.
`strcmp`	Compare strings.
`strmatch`	Find possible matches for a string.

You can also use the following `set` functions with cell arrays of strings.

Function	Description
`intersect`	Set the intersection of two vectors.
`ismember`	Detect members of a set.
`setdiff`	Return the set difference of two vectors.
`setxor`	Set the exclusive OR of two vectors.
`union`	Set the union of two vectors.
`unique`	Set the unique elements of a vector.

Formatting Strings

The following MATLAB functions offer the capability to compose a string that includes ordinary text and data formatted to your specification:

- `sprintf` — Write formatted data to an output string
- `fprintf` — Write formatted data to an output file or the Command Window
- `warning` — Display formatted data in a warning message
- `error` — Display formatted data in an error message and abort
- `assert` — Generate an error when a condition is violated

The syntax of each of these functions includes formatting operators similar to those used by the `printf` function in the C programming language. For example, `%s` tells MATLAB to interpret an input value as a string, and `%d` means to format an integer using decimal notation.

The general formatting syntax for these functions is

```
functionname(..., format_string, value1, value2, ..., valueN)
```

where the `format_string` argument expresses the basic content and formatting of the final output string, and the `value` arguments that follow supply data values to be inserted into the string.

Here is a sample `sprintf` statement, also showing the resulting output string:

```
sprintf('The price of %s on %d/%d/%d was $%.2f.', ...
        'bread', 7, 1, 2006, 2.49)
ans =
    The price of bread on 7/1/2006 was $2.49.
```

The following sections cover

- "The Format String" on page 1-47
- "Input Value Arguments" on page 1-48
- "The Formatting Operator" on page 1-50
- "Constructing the Formatting Operator" on page 1-50
- "Setting Field Width and Precision" on page 1-56
- "Restrictions for Using Identifiers" on page 1-58

Note The examples in this section of the documentation use only the `sprintf` function to demonstrate how string formatting works. However, you can run the examples using the `fprintf`, `warning`, and `error` functions as well.

The Format String

The first input argument in the `sprintf` statement shown above is the `format_string`:

```
'The price of %s on %d/%d/%d was $%.2f.'
```

This argument can include ordinary text, formatting operators and, in some cases, special characters. The formatting operators for this particular string are: `%s`, `%d`, `%d`, `%d`, and `%.2f`.

Following the `format_string` argument are five additional input arguments, one for each of the formatting operators in the string:

```
'bread', 7, 1, 2006, 2.49
```

When MATLAB processes the format string, it replaces each operator with one of these input values.

Special Characters. Special characters are a part of the text in the string. But, because they cannot be entered as ordinary text, they require a unique character sequence to represent them. Use any of the following character sequences to insert special characters into the output string.

To Insert a . . .	Use . . .
Single quotation mark	`''`
Percent character	`%%`
Backslash	`\\`
Backspace	`\b`
Form feed	`\f`
New line	`\n`
Carriage return	`\r`
Horizontal tab	`\t`
Hexadecimal number, `N`	`\xN`
Octal number, `N`	`\N`

Input Value Arguments

In the syntax

```
functionname(..., format_string, value1, value2, ..., valueN)
```

The value arguments must immediately follow format_string in the argument list. In most instances, you supply one of these value arguments for each formatting operator used in the format_string. Scalars, vectors, and numeric and character arrays are valid value arguments. You cannot use cell arrays or structures.

If you include fewer formatting operators than there are values to insert, MATLAB reuses the operators on the additional values. This example shows two formatting operators and six values to insert into the string:

```
sprintf('%s = %d\n', 'A', 479, 'B', 352, 'C', 651)
ans =
    A = 479
    B = 352
    C = 651
```

You can also specify multiple value arguments as a vector or matrix. The format_string needs one %s operator for the entire matrix or vector:

```
mvec = [77 65 84 76 65 66];

sprintf('%s ', char(mvec))
ans =
    MATLAB
```

Sequential and Numbered Argument Specification.

You can place value arguments in the argument list either sequentially (that is, in the same order in which their formatting operators appear in the string), or by identifier (adding a number to each operator that identifies which value argument to replace it with). By default, MATLAB uses sequential ordering.

To specify arguments by a numeric identifier, add a positive integer followed by a $ sign immediately after the % sign in the operator. Numbered argument specification is explained in more detail under the topic "Value Identifiers" on page 1-55.

Ordered Sequentially	Ordered By Identifier

`sprintf('%s %s %s', ...`
`    '1st', '2nd', '3rd')`
`ans =`
`    1st 2nd 3rd`
	`sprintf('%3$s %2$s %1$s', ...`
	`    '1st', '2nd', '3rd')`
	`ans =`
	`     3rd 2nd 1st`

The Formatting Operator

Formatting operators tell MATLAB how to format the numeric or character value arguments and where to insert them into the string. These operators control the notation, alignment, significant digits, field width, and other aspects of the output string.

A formatting operator begins with a `%` character, which may be followed by a series of one or more numbers, characters, or symbols, each playing a role in further defining the format of the insertion value. The final entry in this series is a single *conversion character* that MATLAB uses to define the notation style for the inserted data. Conversion characters used in MATLAB are based on those used by the `printf` function in the C programming language.

Here is a simple example showing five formatting variations for a common value:

```
A = pi*100*ones(1,5);

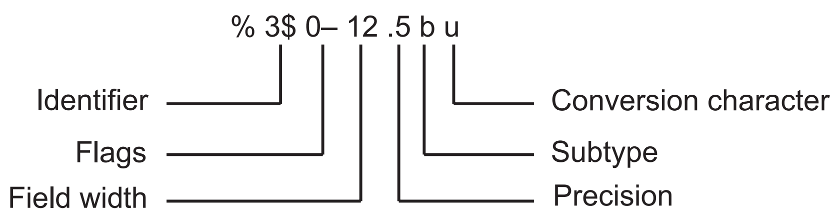
sprintf(' %f \n %.2f \n %+.2f \n %12.2f \n %012.2f \n', A)
ans =
    314.159265      % Display in fixed-point notation (%f)
    314.16          % Display 2 decimal digits (%.2f)
    +314.16         % Display + for positive numbers (%+.2f)
          314.16    % Set width to 12 characters (%12.2f)
    000000314.16    % Replace leading spaces with 0 (%012.2f)
```

Constructing the Formatting Operator

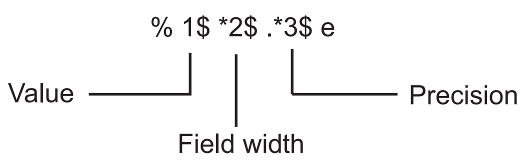
The fields that make up a formatting operator in MATLAB are as shown here, in the order they appear from right to left in the operator. The rightmost field, the conversion character, is required; the five to the left of that are optional. Each of these fields is explained in a section below:

- Conversion Character — Specifies the notation of the output.
- Subtype — Further specifies any nonstandard types.
- Precision — Sets the number of digits to display to the right of the decimal point.
- Field Width — Sets the minimum number of digits to display.
- Flags — Controls the alignment, padding, and inclusion of plus or minus signs.
- Value Identifiers — Map formatting operators to value input arguments. Use the identifier field when value arguments are not in a sequential order in the argument list.

Here is an example of a formatting operator that uses all six fields. (Space characters are not allowed in the operator. They are shown here only to improve readability of the figure).



An alternate syntax, that enables you to supply values for the field width and precision fields from values in the argument list, is shown below. See the section "Specifying Field Width and Precision Outside the format String" on page 1-57 for information on when and how to use this syntax. (Again, space characters are shown only to improve readability of the figure.)



Each field of the formatting operator is described in the following sections. These fields are covered as they appear going from right to left in the

formatting operator, starting with the `Conversion Character` and ending with the `Identifier` field.

Conversion Character. The conversion character specifies the notation of the output. It consists of a single character and appears last in the format specifier. It is the only required field of the format specifier other than the leading `%` character.

Specifier	Description
`c`	Single character
`d`	Decimal notation (signed)
`e`	Exponential notation (using a lowercase `e` as in `3.1415e+00`)
`E`	Exponential notation (using an uppercase `E` as in `3.1415E+00`)
`f`	Fixed-point notation
`g`	The more compact of `%e` or `%f`. (Insignificant zeros do not print.)
`G`	Same as `%g`, but using an uppercase `E`
`o`	Octal notation (unsigned)
`s`	String of characters
`u`	Decimal notation (unsigned)
`x`	Hexadecimal notation (using lowercase letters `a`–`f`)
`X`	Hexadecimal notation (using uppercase letters `A`–`F`)

This example uses conversion characters to display the number 46 in decimal, fixed-point, exponential, and hexadecimal formats:

```
A = 46*ones(1,4);

sprintf('%d   %f   %e   %X', A)
ans =
    46   46.000000   4.600000e+001   2E
```

Subtype. The subtype field is a single alphabetic character that immediately precedes the conversion character. The following nonstandard subtype specifiers are supported for the conversion characters `%o`, `%u`, `%x`, and `%X`.

`b`	The underlying C data type is a double rather than an unsigned integer. For example, to print a double-precision value in hexadecimal, use a format like `'%bx'`.
`t`	The underlying C data type is a float rather than an unsigned integer.

Precision. `precision` in a formatting operator is a nonnegative integer that tells MATLAB how many digits to the right of the decimal point to use when formatting the corresponding input value. The `precision` field consists of a nonnegative integer that immediately follows a period and extends to the first alphabetic character after that period. For example, the specifier `%7.3f`, has a precision of 3.

Here are some examples of how the `precision` field affects different types of notation:

```
sprintf('%g   %.2g   %f   %.2f', pi*50*ones(1,4))
ans =
    157.08   1.6e+002   157.079633   157.08
```

Precision is not usually used in format specifiers for strings (i.e., `%s`). If you do use it on a string and if the value `p` in the precision field is less than the number of characters in the string, MATLAB displays only `p` characters of the string and truncates the rest.

You can also supply the value for a precision field from outside of the format specifier. See the section “Specifying Field Width and Precision Outside the format String” on page 1-57 for more information on this.

For more information on the use of `precision` in formatting, see “Setting Field Width and Precision” on page 1-56.

Field Width. `Field width` in a formatting operator is a nonnegative integer that tells MATLAB the minimum number of digits or characters to use when formatting the corresponding input value. For example, the specifier `%7.3f`, has a width of 7.

Here are some examples of how the `width` field affects different types of notation:

```
sprintf('|%e|%15e|%f|%15f|', pi*50*ones(1,4))
ans =
    |1.570796e+002|  1.570796e+002|157.079633|     157.079633|
```

When used on a string, the `field width` can determine whether MATLAB pads the string with spaces. If `width` is less than or equal to the number of characters in the string, it has no effect.

```
sprintf('%30s', 'Pad left with spaces')
ans =
          Pad left with spaces
```

You can also supply a value for `field width` from outside of the format specifier. See the section "Specifying Field Width and Precision Outside the format String" on page 1-57 for more information on this.

For more information on the use of `field width` in formatting, see "Setting Field Width and Precision" on page 1-56.

Flags. You can control the alignment of the output using any of these optional flags:

Character	Description	Example
A minus sign (-)	Left-justifies the converted argument in its field	`%-5.2d`
A plus sign (+)	Always prints a sign character (+ or –)	`%+5.2d`
Zero (0)	Pad with zeros rather than spaces.	`%05.2f`

Right- and left-justify the output. The default is to right-justify:

```
sprintf('right-justify: %12.2f\nleft-justify: %-12.2f', ...
        12.3, 12.3)
ans =
    right-justify:        12.30
    left-justify: 12.30
```

Display a + sign for positive numbers. The default is to omit the + sign:

```
sprintf('no sign: %12.2f\nsign: %+12.2f', ...
        12.3, 12.3)
ans =
    no sign:        12.30
    sign:       +12.30
```

Pad to the left with spaces or zeros. The default is to use space-padding:

```
sprintf('space-padded: %12.2f\nzero-padded: %012.2f', ...
        5.2, 5.2)
ans =
    space-padded:         5.20
    zero-padded: 000000005.20
```

Note You can specify more than one flag in a formatting operator.

Value Identifiers. By default, MATLAB inserts data values from the argument list into the string in a sequential order. If you have a need to use the value arguments in a nonsequential order, you can override the default by using a numeric identifier in each format specifier. Specify nonsequential arguments with an integer immediately following the % sign, followed by a $ sign.

Ordered Sequentially	Ordered By Identifier

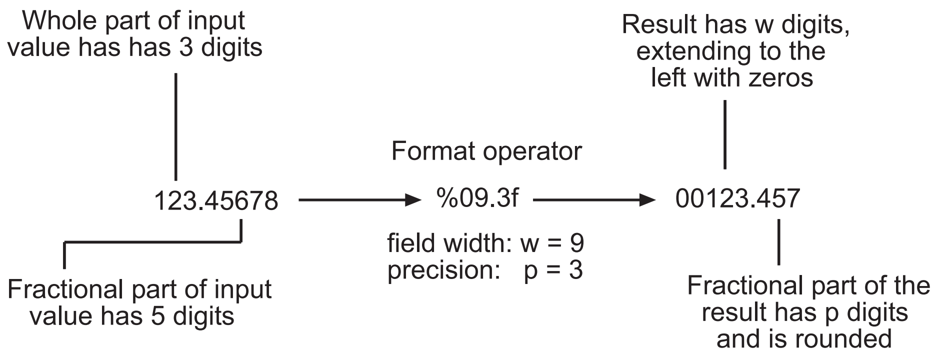
`sprintf('%s %s %s', ...`
`    '1st', '2nd', '3rd')`
`ans =`
`    1st 2nd 3rd`
	`sprintf('%3$s %2$s %1$s', ...`
	`    '1st', '2nd', '3rd')`
	`ans =`
	`    3rd 2nd 1st`

Setting Field Width and Precision

This section provides further information on the use of the field width and precision fields of the formatting operator:

- "Effect on the Output String" on page 1-56
- "Specifying Field Width and Precision Outside the format String" on page 1-57
- "Using Identifiers In the Width and Precision Fields" on page 1-57

Effect on the Output String. The figure below illustrates the way in which the field width and precision settings affect the output of the string formatting functions. In this figure, the zero following the `%` sign in the formatting operator means to add leading zeros to the output string rather than space characters:



General rules for formatting

- If precision is not specified, it defaults to 6.
- If precision (`p`) is less than the number of digits in the fractional part of the input value (`f`), then only `p` digits are shown to the right of the decimal point in the output, and that fractional value is rounded.
- If precision (`p`) is greater than the number of digits in the fractional part of the input value (`f`), then `p` digits are shown to the right of the decimal point in the output, and the fractional part is extended to the right with `p-f` zeros.

- If field width is not specified, it defaults to `precision` + 1 + the number of digits in the whole part of the input value.
- If field width (`w`) is greater than `p+1` plus the number of digits in the whole part of the input value (`n`), then the whole part of the output value is extended to the left with `w-(n+1+p)` space characters or zeros, depending on whether or not the `zero` flag is set in the `Flags` field. The default is to extend the whole part of the output with space characters.

Specifying Field Width and Precision Outside the format String. To specify field width or precision using values from a sequential argument list, use an asterisk (`*`) in place of the `field width` or `precision` field of the formatting operator.

This example formats and displays three numbers. The formatting operator for the first, `%*f`, has an asterisk in the field width location of the formatting operator, specifying that just the field width, 15, is to be taken from the argument list. The second operator, `%.*f` puts the asterisk after the decimal point meaning, that it is the precision that is to take its value from the argument list. And the third operator, `%*.*f`, specifies both field width and precision in the argument list:

```
 sprintf('%*f   %.*f   %*.*f', ...
       15, 123.45678, ...     % Width for 123.45678 is 15
       3, 16.42837, ...       % Precision for rand*20 is .3
       6, 4, pi)              % Width & Precision for pi is 6.4
ans =
     123.456780   16.428   3.1416
```

You can mix the two styles. For example, this statement gets the field width from the argument list and the precision from the format string:

```
sprintf('%*.2f', 5, 123.45678)
ans =
    123.46
```

Using Identifiers In the Width and Precision Fields. You can also derive field width and precision values from a nonsequential (i.e., numbered) argument list. Inside the formatting operator, specify `field width` and/or `precision` with an asterisk followed by an identifier number, followed by a `$` sign.

This example from the previous section shows how to obtain field width and precision from a sequential argument list:

```
sprintf('%*f   %.*f   %*.*f', ...
        15, 123.45678, ...
        3, 16.42837, ...
        6, 4, pi)

ans =
     123.456780   16.428   3.1416
```

Here is an example of how to do the same thing using numbered ordering. Field width for the first output value is 15, precision for the second value is 3, and field width and precision for the third value is 6 and 4, respectively. If you specify field width or precision with identifiers, then you must specify the value with an identifier as well:

```
sprintf('%1$*4$f   %2$.*5$f   %3$*6$.*7$f', ...
123.45678, 16.42837, pi, 15, 3, 6, 4)

ans =
     123.456780   16.428   3.1416
```

Restrictions for Using Identifiers

If any of the formatting operators in a string include an identifier field, then all of the operators in that string must do the same; you cannot use both sequential and nonsequential ordering in the same function call.

Valid Syntax	Invalid Syntax

`sprintf('%d %d %d %d', ...`
`        1, 2, 3, 4)`
`ans =`
`    1 2 3 4`
	`sprintf('%d %3$d %d %d', ...`
	`        1, 2, 3, 4)`
	`ans =`
	`    1`

If your command provides more value arguments than there are formatting operators in the format string, MATLAB reuses the operators. However, MATLAB allows this only for commands that use sequential ordering.

You cannot reuse formatting operators when making a function call with numbered ordering of the value arguments.

Valid Syntax	Invalid Syntax

`sprintf('%d', 1, 2, 3, 4)`
`ans =`
`    1234`
	`sprintf('%1$d', 1, 2, 3, 4)`
	`ans =`
	`    1`

Also, do not use identifiers when the value arguments are in the form of a vector or array:

Valid Syntax	Invalid Syntax

`v = [1.4 2.7 3.1];`
 	
`sprintf('%.4f %.4f %.4f', v)`
`ans =`
`    1.4000 2.7000 3.1000`
	`v = [1.4 2.7 3.1];`
	
	`sprintf('%3$.4f %1$.4f %2$.4f', v)`
	`ans =`
	`   Empty string: 1-by-0`

String Comparisons

There are several ways to compare strings and substrings:

- You can compare two strings, or parts of two strings, for equality.
- You can compare individual characters in two strings for equality.
- You can categorize every element within a string, determining whether each element is a character or white space.

These functions work for both character arrays and cell arrays of strings.

Comparing Strings for Equality

You can use any of four functions to determine if two input strings are identical:

- `strcmp` determines if two strings are identical.

- `strncmp` determines if the first `n` characters of two strings are identical.
- `strcmpi` and `strncmpi` are the same as `strcmp` and `strncmp`, except that they ignore case.

Consider the two strings

```
str1 = 'hello';
str2 = 'help';
```

Strings `str1` and `str2` are not identical, so invoking `strcmp` returns logical `0` (`false`). For example,

```
C = strcmp(str1,str2)
C =
     0
```

Note For C programmers, this is an important difference between the MATLAB `strcmp` and C `strcmp()`functions, where the latter returns 0 if the two strings are the same.

The first three characters of `str1` and `str2` are identical, so invoking `strncmp` with any value up to 3 returns `1`:

```
C = strncmp(str1, str2, 2)
C =
     1
```

These functions work cell-by-cell on a cell array of strings. Consider the two cell arrays of strings

```
A = {'pizza'; 'chips'; 'candy'};
B = {'pizza'; 'chocolate'; 'pretzels'};
```

Now apply the string comparison functions:

```
strcmp(A,B)
ans =
     1
     0
```

```
    0
strncmp(A,B,1)
ans =
    1
    1
    0
```

Comparing for Equality Using Operators

You can use MATLAB relational operators on character arrays, as long as the arrays you are comparing have equal dimensions, or one is a scalar. For example, you can use the equality operator (==) to determine where the matching characters are in two strings:

```
A = 'fate';
B = 'cake';

A == B
ans =
    0    1    0    1
```

All of the relational operators (>, >=, <, <=, ==, ~=) compare the values of corresponding characters.

Categorizing Characters Within a String

There are three functions for categorizing characters inside a string:

1. `isletter` determines if a character is a letter.
2. `isspace` determines if a character is white space (blank, tab, or new line).
3. `isstrprop` checks characters in a string to see if they match a category you specify, such as
   - Alphabetic
   - Alphanumeric
   - Lowercase or uppercase
   - Decimal digits
   - Hexadecimal digits

- Control characters
- Graphic characters
- Punctuation characters
- Whitespace characters

For example, create a string named `mystring`:

```
mystring = 'Room 401';
```

`isletter` examines each character in the string, producing an output vector of the same length as `mystring`:

```
A = isletter(mystring)
A =
     1    1    1    1    0    0    0    0
```

The first four elements in `A` are logical `1` (`true`) because the first four characters of `mystring` are letters.

Searching and Replacing

MATLAB provides several functions for searching and replacing characters in a string. (MATLAB also supports search and replace operations using regular expressions. See Regular Expressions.)

Consider a string named `label`:

```
label = 'Sample 1, 10/28/95';
```

The `strrep` function performs the standard search-and-replace operation. Use `strrep` to change the date from `'10/28'` to `'10/30'`:

```
newlabel = strrep(label, '28', '30')
newlabel =
   Sample 1, 10/30/95
```

`findstr` returns the starting position of a substring within a longer string. To find all occurrences of the string `'amp'` inside `label`, use

```
position = findstr('amp', label)
```

```
position =
    2
```

The position within `label` where the only occurrence of `'amp'` begins is the second character.

The `strtok` function returns the characters before the first occurrence of a delimiting character in an input string. The default delimiting characters are the set of white-space characters. You can use the `strtok` function to parse a sentence into words. For example,

```
function allWords = words(inputString)
remainder = inputString;
allWords = '';

while (any(remainder))
  [chopped,remainder] = strtok(remainder);
  allWords = strvcat(allWords, chopped);
end
```

You can also use the `textscan` function to perform this task.

The `strmatch` function looks through the rows of a character array or cell array of strings to find strings that begin with a given series of characters. It returns the indices of the rows that begin with these characters:

```
maxstrings = strvcat('max', 'minimax', 'maximum')
maxstrings =
   max
   minimax
   maximum

strmatch('max', maxstrings)
ans =
     1
     3
```

Converting from Numeric to String

The functions listed in this table provide a number of ways to convert numeric data to character strings.

Function	Description	Example
`char`	Convert a positive integer to an equivalent character. (Truncates any fractional parts.)	`[72 105]` → `'Hi'`
`int2str`	Convert a positive or negative integer to a character type. (Rounds any fractional parts.)	`[72 105]` → `'72 105'`
`num2str`	Convert a numeric type to a character type of the specified precision and format.	`[72 105]` → `'72/105/'` (format set to `%1d/`)
`mat2str`	Convert a numeric type to a character type of the specified precision, returning a string MATLAB can evaluate.	`[72 105]` → `'[72 105]'`
`dec2hex`	Convert a positive integer to a character type of hexadecimal base.	`[72 105]` → `'48 69'`
`dec2bin`	Convert a positive integer to a character type of binary base.	`[72 105]` → `'1001000 1101001'`
`dec2base`	Convert a positive integer to a character type of any base from 2 through 36.	`[72 105]` → `'110 151'` (base set to `8`)

Converting to a Character Equivalent

The `char` function converts integers to Unicode character codes and returns a string composed of the equivalent characters:

```
x = [77 65 84 76 65 66];
char(x)
ans =
   MATLAB
```

Converting to a String of Numbers

The `int2str`, `num2str`, and `mat2str` functions convert numeric values to strings where each character represents a separate digit of the input value. The `int2str` and `num2str` functions are often useful for labeling plots. For example, the following lines use `num2str` to prepare automated labels for the *x*-axis of a plot:

```
function plotlabel(x, y)
```

```
plot(x, y)
str1 = num2str(min(x));
str2 = num2str(max(x));
out = ['Value of f from ' str1 ' to ' str2];
xlabel(out);
```

Converting to a Specific Radix

Another class of conversion functions changes numeric values into strings representing a decimal value in another base, such as binary or hexadecimal representation. This includes `dec2hex`, `dec2bin`, and `dec2base`.

Converting from String to Numeric

The functions listed in this table provide a number of ways to convert character strings to numeric data.

Function	Description	Example
`uintN` (e.g., `uint8`)	Convert a character to an integer code that represents that character.	`'Hi'` → `72 105`
`str2num`	Convert a character type to a numeric type.	`'72 105'` → `[72 105]`
`str2double`	Similar to `str2num`, but offers better performance and works with cell arrays of strings.	`{'72' '105'}` → `[72 105]`
`hex2num`	Convert a numeric type to a character type of specified precision, returning a string that MATLAB can evaluate.	`'A'` → `'-1.4917e-154'`
`hex2dec`	Convert a character type of hexadecimal base to a positive integer.	`'A'` → `10`
`bin2dec`	Convert a positive integer to a character type of binary base.	`'1010'` → `10`
`base2dec`	Convert a positive integer to a character type of any base from 2 through 36.	`'12'` → `10` (if `base == 8`)

Converting from a Character Equivalent

Character arrays store each character as a 16-bit numeric value. Use one of the integer conversion functions (e.g., `uint8`) or the `double` function to convert strings to their numeric values, and `char` to revert to character representation:

```
name = 'Thomas R. Lee';

name = double(name)
name =
    84  104  111  109  97  115  32  82  46  32  76  101  101

name = char(name)
name =
   Thomas R. Lee
```

Converting from a Numeric String

Use `str2num` to convert a character array to the numeric value represented by that string:

```
str = '37.294e-1';

val = str2num(str)
val =
    3.7294
```

The `str2double` function converts a cell array of strings to the double-precision values represented by the strings:

```
c = {'37.294e-1'; '-58.375'; '13.796'};

d = str2double(c)
d =
    3.7294
  -58.3750
   13.7960

whos
  Name      Size                   Bytes  Class
```

```
c          3x1                        224  cell
d          3x1                         24  double
```

Converting from a Specific Radix

To convert from a character representation of a nondecimal number to the value of that number, use one of these functions: `hex2num`, `hex2dec`, `bin2dec`, or `base2dec`.

The `hex2num` and `hex2dec` functions both take hexadecimal (base 16) inputs, but `hex2num` returns the IEEE double-precision floating-point number it represents, while `hex2dec` converts to a decimal integer.

Function Summary

MATLAB provides these functions for working with character arrays:

- Functions to Create Character Arrays on page 1-67
- Functions to Modify Character Arrays on page 1-68
- Functions to Read and Operate on Character Arrays on page 1-68
- Functions to Search or Compare Character Arrays on page 1-68
- Functions to Determine Class or Content on page 1-69
- Functions to Convert Between Numeric and String Classes on page 1-69
- Functions to Work with Cell Arrays of Strings as Sets on page 1-69

Functions to Create Character Arrays

Function	Description
'str'	Create the string specified between quotes.
`blanks`	Create a string of blanks.
`sprintf`	Write formatted data to a string.
`strcat`	Concatenate strings.
`strvcat`	Concatenate strings vertically.

Functions to Modify Character Arrays

Function	Description
`deblank`	Remove trailing blanks.
`lower`	Make all letters lowercase.
`sort`	Sort elements in ascending or descending order.
`strjust`	Justify a string.
`strrep`	Replace one string with another.
`strtrim`	Remove leading and trailing white space.
`upper`	Make all letters uppercase.

Functions to Read and Operate on Character Arrays

Function	Description
`eval`	Execute a string with MATLAB expression.
`sscanf`	Read a string under format control.

Functions to Search or Compare Character Arrays

Function	Description
`findstr`	Find one string within another.
`strcmp`	Compare strings.
`strcmpi`	Compare strings, ignoring case.
`strmatch`	Find matches for a string.
`strncmp`	Compare the first N characters of strings.
`strncmpi`	Compare the first N characters, ignoring case.
`strtok`	Find a token in a string.

Functions to Determine Class or Content

Function	Description
iscellstr	Return true for acell array of strings.
ischar	Return true for a character array.
isletter	Return true for letters of the alphabet.
isstrprop	Determine if a string is of the specified category.
isspace	Return true for white-space characters.

Functions to Convert Between Numeric and String Classes

Function	Description
char	Convert to a character or string.
cellstr	Convert a character array to a cell array of strings.
double	Convert a string to numeric codes.
int2str	Convert an integer to a string.
mat2str	Convert a matrix to a string you can run eval on.
num2str	Convert a number to a string.
str2num	Convert a string to a number.
str2double	Convert a string to a double-precision value.

Functions to Work with Cell Arrays of Strings as Sets

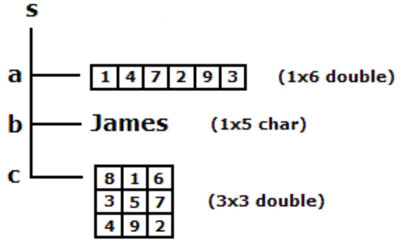
Function	Description
intersect	Set the intersection of two vectors.
ismember	Detect members of a set.
setdiff	Return the set difference of two vectors.
setxor	Set the exclusive OR of two vectors.
union	Set the union of two vectors.
unique	Set the unique elements of a vector.

Structures

In this section...

What Is a Structure?

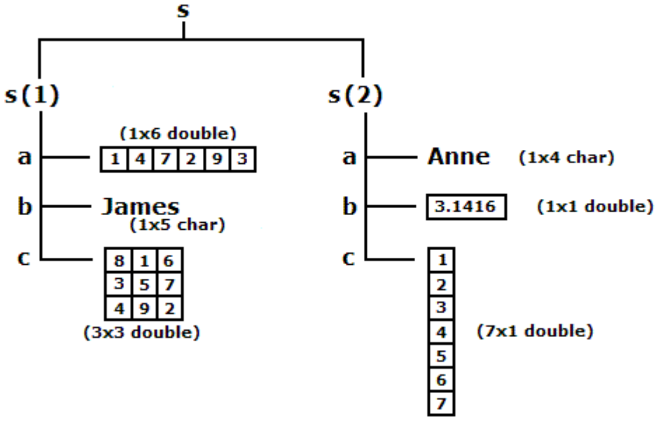
A structure is a MATLAB data type that provides the means to store selected data together in a single entity. A structure consists mainly of data containers, called *fields*, and each of these fields stores an array of some MATLAB data type. You assign a name to each field as you create the structure. The figure below shows a structure `s` that has three fields: `a`, `b`, and `c`.



Each field of a structure contains a separate MATLAB array. This array can belong to any one MATLAB or user-defined class, and can have any valid

array dimensions. A second field of the same structure can belong to an entirely different class, and can also have different dimensions than the first. The fields of the structure shown above, for example, contain a 1-by-6 array of class `double`, a 1-by-5 array of class `char`, and a 3-by-3 array of `double`.

Like all MATLAB data types, the structure is an array. The class of a structure is called *struct*, so an array of structures is often referred to as a struct array. Like other MATLAB arrays, a struct array can have any dimensions. The struct array shown below has the dimensions 1-by-2 and is composed of two elements: `s(1)` and `s(2)`. Each of these elements is a structure with fields `a`, `b`, and `c` of its own.



Each element of a struct array must contain the same number of fields and use the same field names as every other element of that struct array. The arrays of data that are stored in these fields, however, do not have to match; they can belong to different classes, and they can have different dimensions. In the struct array shown above, for example, fields `b` and `c` of element `s(1)` contain arrays of different classes and dimensions. The same holds true for fields that are named the same but belong to different elements of the struct array. An example of this is field `b` in `s(1)` and field `b` in `s(2)`.

Reasons to Use a Structure

Perhaps the most common reason for using a structure (or a cell array) is the ability to store arrays of mixed classes and sizes. Most MATLAB arrays must contain the same number of elements which must also be of the same class. The role of the structure, and cell array, as containers of heterogeneous data is very important.

A structure also provides the means to store selected data together in a single entity. This offers you the ability to access and operate on all or parts of the data collectively. You can apply functions directly to a structure, pass the structure to and from your M-file functions, display the value of any fields, or perform most any MATLAB operation on the contents of a structure.

A third reason to use structures is that they give you the ability to apply text labels to your data, as opposed to using array subscripting.

Comparing Struct Arrays with Cell Arrays

Struct arrays and cell arrays are similar in purpose and, in some ways, also design. Both provide a means of storing heterogeneous data in containers of different size and type. Probably the most noticeable *difference* between the two is that the containers of a structure are named fields, whereas a cell array uses numerically indexed cells.

Structures are often used in applications where organization of the data is of high importance. Cell arrays are often used when working with data that you intend to process by index in a programming control loop. Cell arrays are also useful in storing character strings of different lengths.

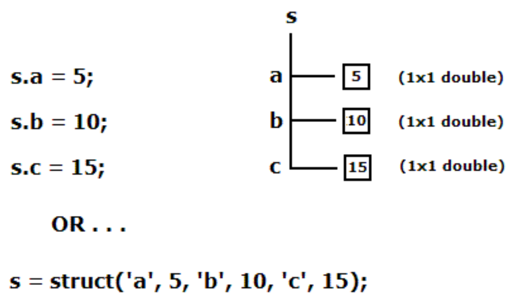
There are many reasons for choosing either structures, cell arrays, or both for your work in MATLAB. For more information on cell arrays, see “Cell Arrays” on page 1-101 in the MATLAB Programming Fundamentals documentation.

Creating a Structure

This section describes how to create struct arrays of different shapes and sizes, how MATLAB keeps the number of fields the same for all elements, and how to preallocate memory for larger structures.

Creating Structures and Structure Fields

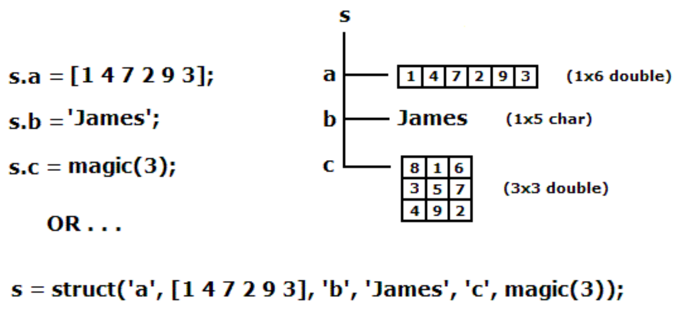
There are two ways of creating a MATLAB structure: by individual assignments to its fields, or by a single call to the `struct` function. The figure below shows a 1-by-1 structure `s` having three fields: `a`, `b`, and `c`. The two sets of commands that can create this structure are shown to the left of and below the figure:



The three statements at the top left are an example of individual assignment to fields. The syntax `s.a` used in the first of these statements refers to field `a` of a structure `s`. The structure does not exist yet, so MATLAB creates the structure and the given field `a`, and assigns the value 5 to the field. The two remaining statements add two new fields to the structure and set their values to 10 and 15, respectively.

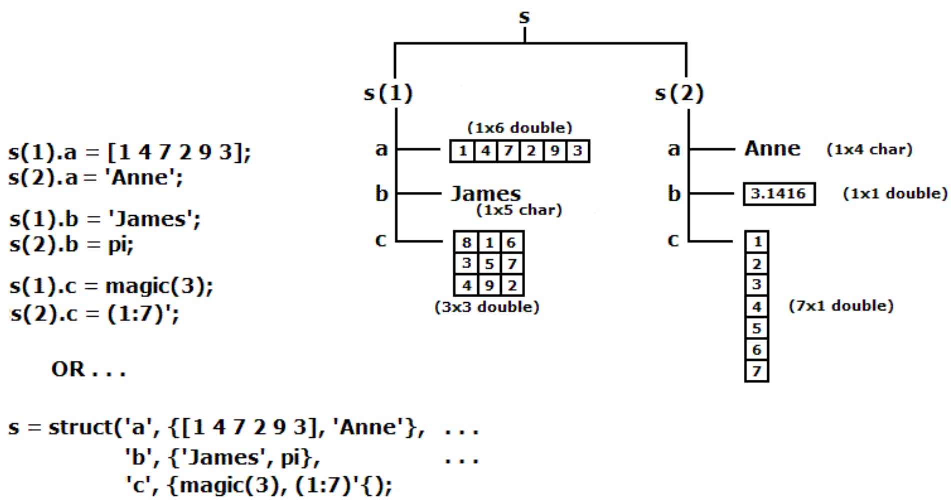
The statement at the bottom left uses the `struct` function to create the structure and its fields and to assign the respective values to each. See the reference page for the `struct` function for the various syntax options you can use.

Structures with Nonscalar Fields. The next figure creates a three-field structure to contain *nonscalar* array values. The statements used to create this structure are very much like those used in the previous figure.



Nonscalar Struct Arrays. The next figure creates a struct array `s` that has two elements, `s(1)` and `s(2)`. Each element of the struct array has three fields: `a`, `b`, and `c`. To create this array by assigning to fields, you need to specify which element of `s` you are assigning to. For example, to create a field `b` for element 2 of struct array `s`, assign to `s(2).b`.

The `struct` function requires a slightly different syntax when creating fields in multiple elements of the struct array. Follow each field name argument with a list of values enclosed in curly braces `{}`. The enclosed list specifies the values to be assigned to that field for the successive elements of the struct array. For example, the first two input arguments shown in the `struct` command below are `'a'` and `{[1 4 7 2 9 3],'Anne'}`. This tells MATLAB to assign the vector `[1 4 7 2 9 3]` to `s(1).a`, and the string `'Anne'` to `s(2).a`:



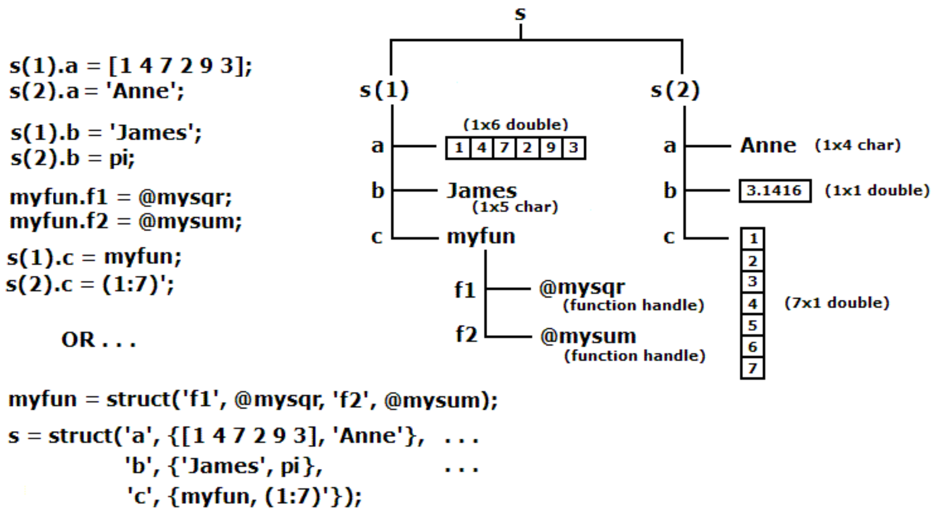
In MATLAB, curly braces `{}` operator constructs a cell array. One use of cell arrays is as a convenient way to pass arguments when calling a function. This is exactly how they are used with the `struct` function. When you use this operator to pass multiple field values to the `struct` function, you are actually passing these values packaged in a cell array. The `struct` function, upon receiving the cell array argument, removes the field values from the cell array and assigns them to the fields specified in your `struct` command.

Suppose that, in the example above, you want to create a field `a` of for struct element `s(1)`. But instead of `s(1).a` being a 1-by-6 numeric array, you want it to be a 1-by-6 cell array. In that case, you would need to enclose the first argument itself with curly braces:

```
s = struct('a', {{1 4 7 2 9 3}, 'Anne'}, ...
           'b', {'James', pi}, ...
           'c', {magic(3), (1:7)'});
```

Note When calling the `struct` function, use one set of curly braces `{}` to pass multiple field values, and use two sets of curly braces `{{}}` to create a cell array in the specified field.

Nested Structures. The next figure shows an example of one struct array stored (or *nested*) within another struct array. The inner struct array, called `myfun`, is a collection of two function handles. The commands shown to the left build the two structures, storing the inner structure in field `c` of the first element of the outer structure:



See the reference page for the `struct` function for more information on using this function to create structures.

Handling Unassigned Fields

Each element of a nonscalar struct array must have the same set of fields. Whenever you add a new field to a struct array, MATLAB adds a field of the same name to all elements of the struct array.

For example, if you enter the following commands:

```
s.a = 5;
s.b = 10;
s(2).a = 15;
```

MATLAB creates and assigns values to the three specified fields, and also creates a unspecified field `s(2).b`, setting its value to the empty array (`[]`). This ensures that the fields of `s(1)` and `s(2)` are the same in number and name. This is called *scalar expansion*:

```
s(2).b
ans =
     []
```

MATLAB also automatically keeps field naming and count the same for all elements when you use the `struct` function to create a struct array. In this case, however, field `b` of elements `s(2)` and `s(3)` are set to the value specified for `s(1).b`, which is 10:

```
s = struct('a', {5, 15, 25}, 'b', 10);
[v1 v2 v3] = s.b;

[v1  v2  v3]
ans =
    10    10    10
```

Note The number of field values expressed in curly braces must be the same for each field name with the exception of fields that are scalar, in which case you do not need the curly braces.

This example calls `struct` with three values for field `a` and two values for field `b`, causing the command to generate an error:

```
s = struct('a', {5, 15, 25}, 'b', {10, 15});
??? Error using ==> struct
Array dimensions of input 4 must match those of input 2 or be scalar.
```

Preallocating Memory for the Array

MATLAB stores the field names and any overhead information required by a struct array in contiguous memory. If you increase the number of fields used by a struct array over time, even if this happens just by increasing the dimensions of the array, MATLAB uses up more of this contiguous piece of

memory for field name storage. This can eventually lead to “out of memory” errors.

If you can roughly estimate the number of fields and the number of struct array elements at the time you create a struct array, you can preallocate the necessary space in memory and help to avoid this type of problem. See the documentation on Data Structures and Memory to help you make this estimate.

Note Unlike the field name and internal header information of a struct array, the memory consumed by the *data* stored in a struct array is not contiguous. While preallocating memory can help avoid memory problems when increasing the dimensions of a struct array, it does not protect against shortages in memory due to the amount of data you store in the array. Even with preallocated struct (and cell) arrays, you need to take precautions against using more memory than is available.

How to Preallocate Memory. Methods for preallocating and initializing a struct array are as follows:

- To allocate memory for an 25-by-50 struct array with fields `a`, `b`, and `c` and initialize the entire array to `[]`, use either of the following two methods:

  ```
  S(25,50) = struct( 'a', [], 'b', [], 'c',[]);
  S(25,50).a = [];   S(25,50).b = [];   S(25,50).c = [];
  ```

- To allocate memory for the same struct array, initializing the fields of one element as specified, and copying that element to all elements of the struct array `S`, use either of the following two methods:

  ```
  S(1:25,1:50) = struct('a', 20, 'b', 30, 'c', 40);
  S = repmat(struct('a', 20, 'b', 30, 'c', 40), [25 50]);
  ```

After the memory has been allocated, you can begin to construct the array by assigning data to it.

Structure Fields

This section describes how to name the fields you create, how to find out what fields a structure contains, how to create and assign field names at run time, and the functions that MATLAB provides to help work with the fields of a structure.

Guidelines for Naming Structure Fields

A field name is just a character string. MATLAB field names must follow the same rules as standard MATLAB variables:

1. Structure field names must begin with a letter, and can contain additional letters, digits, or underscore characters.
2. It is advisable to keep field names to a maximum of N characters, where N is the number returned by the function `namelengthmax`. MATLAB accepts longer names, but only uses the first N characters and ignores the rest.
3. MATLAB distinguishes between uppercase and lowercase characters. The field name `S.income` is not the same as the name `S.Income`.
4. In most cases, you should refrain from using the names of functions or other active variables as field names.

Listing the Fields of a Structure

To access the contents of a struct array, you first need to find out what the names of its fields are. The `fieldnames` function returns a cell array of strings listing all fields belonging to the structure array. The fields appear in the order in which they were created.

Here is a structure with four fields:

```
USPres.name = 'Franklin D. Roosevelt';
USPres.vp(1) = {'John Garner'};
USPres.vp(2) = {'Henry Wallace'};
USPres.vp(3) = {'Harry S Truman'};
USPres.term = [1933, 1945];
USPres.party = 'Democratic';
```

The `fieldnames` function returns the names of each field of `USPres` in a 4-by-1 cell array of strings:

```
presFields = fieldnames(USPres)
ans =
    'name'
    'vp'
    'term'
    'party'
```

Arranging Fieldnames Alphabetically

The `orderfields` function returns a new struct array that is just like the original, except that the order of the field names is alphabetical. If you assign the output of `orderfields` back to the input structure, it effectively modifies the field ordering of the input structure:

For struct array `USPres`,

```
orderfields(USPres)
ans =
1x32 struct array with fields:
    name
    party
    term
    vp
```

returns the field names of the struct array, listed in alphabetical order. If you want to actually change the order of fields in the struct array, assign the value returned by `orderfields` back to `USPres`:

```
USPres = orderfields(USPres);
```

Creating Field Names Dynamically

Another way to give a name to a structure field is to derive the name at the time MATLAB executes your code. First, establish a variable to represent the field name of your structure. Then, at run time, MATLAB uses the current value of this variable as the field name. This is called a *dynamic field name*. You can only use dynamic field names when you create your struct array using individual assignment to fields.

The syntax for creating a field name dynamically is

```
structName.(dynamicExpression) = fieldValue;
```

The term `dynamicExpression` is any MATLAB expression that returns a character or string. For example, in the following statement, the `datestr` function returns the string `Nov2708` which then becomes a field name in the `price` structure. The dot-parentheses `.( )` syntax tells MATLAB that the string value returned by `datestr(now,'mmmddyy')` is a field name for the structure:

```
price.(datestr(now,'mmmddyy')) = 89.99;
```

Examining the field names for the `price` structure shows the `Nov2708` field just added:

```
fieldnames(price)
ans =
    'Nov2708'

price.Nov0708
ans =
     89.9900
```

Functions That Operate on Fields

The following functions are commonly used with the field names of structures. For more information on these functions, consult the MATLAB Function Reference documentation:

Function	Description	Return Value
`fieldnames`	Get all field names of specified structure.	Cell array of strings listing fields of input structure in the order in which they were assigned to the structure.
`getfield`	Get contents of the specified field.	Current value assigned to specified field.
`isfield`	Determine if input is a structure field.	`true` if the field is a structure field. Otherwise, `false`.

Function	Description	Return Value
`orderfields`	Order fields of a structure array.	Input structure with fields ordered alphabetically.
`rmfield`	Remove structure field.	Input structure with specified field removed.
`setfield`	Set structure field contents, returning the modified structure.	Input structure with specified field set to new value.

To set the value of a structure field, you can either assign it directly, or use the `setfield` function. Likewise, you can obtain the value of a field as shown in the section "Returning Data from a Struct Array" on page 1-84 or by using the `getfield` function.

Indexing into a Struct Array

This section describes how to index into the elements of a struct array, and any arrays that are contained within a struct field.

Basic Struct and Field Indexing

The most general indexing with which to store data into or retrieve data from a struct array is

```
structName(sRows, sCols, ...).fieldName(fRows, fCols, ...)
```

If the structure is scalar, then you can omit the structure indexing as shown here:

```
structName.fieldName(fRows, fCols, ...)
```

Indexing to Inner Levels of the Struct Array

The fields of a structure contain arrays of standard MATLAB data types. These arrays use the indexing syntax appropriate to the class of the array. The table below shows examples of statements that use a combination of struct and cell array indexing:

Array Element	Syntax to Use
Access an element of an array A, where A is a field of structure S.	S(3,15).A(5,25)
Access an element of cell array A, where A is a field of structure S.	S(3,15).A{5,20}
Access an element of array B, where B is a field of struct array A, and A is a field of struct array S.	S(3,15).A(5,20).B(50,5)
Access an element of array B, where B is a field of a structure that resides in cell array A, and A is a field of structure S.	S(3,15).A{5,20}.B(50,5)
Access an element of cell array B, where B is a field of structure A, and A is a field of structure S.	S(3,15).A.B{5,20}

Indexing Tips

Some techniques that could help you in determining how to format struct array indexing are:

- Use the whos function to tell you exactly what the class and size of the variable is that you are dealing with. Combining this information and the standard indexing rules should enable you to find the appropriate syntax to use to get to the desired piece of data.
- Enter only the right side of the assignment statement, effectively assigning to the ans variable. By not confining MATLAB to fit its return data into a possibly incompatible data structure, you allow the software to decide the type and size of array needed to contain this data. In so doing, the output illustrates or implies the type of indexing required.
- There are instances in which you can enter a perfectly good indexing statement that will fail just the same. The reason for this failure is that the variable you are attempting to assign to already exists in the workspace. This variable represents an array that is incompatible with your current assignment statement.

If you are assigning to a variable that is already in use, try clearing the variable from the MATLAB workspace, and then reentering your indexing statement.

- You can index into a nested array in stages rather than all at once. Consider breaking down this indexing expression:

```
S(5,3).A(4,7).B(:,4)
```

into the following:

```
x = S(5,3).A;    % x is a struct array
y = x(4,7).B;    % y is also a struct array
z = y(:,4)       % z is a standard array
```

Returning Data from a Struct Array

The following table shows a number of different ways of returning data from a struct array. The variable `s` is a 3-by-4 structure with fields `a`, `b`, and `c`. Each of these fields is a 2-by-5 array of class `double`:

Values To Be Acquired	MATLAB Statement	Data Structure Returned
The entire struct array `s`	`s`	3x4 struct array with fields `a`, `b`, and `c`.
The entire struct array `s`, as a vector	`s(:)`	12x1 struct array with fields `a`, `b`, and `c`.
Selected elements of struct `s`	`s(2:3,1:3)`	2x3 struct array with fields `a`, `b`, and `c`.
The full array `a` in selected element of `s`.	`s(2,3).a`	2x5 array of `double`.
The full array `a` in multiple elements of `s`.	`s(2:3,3:4).a`	4-item comma-separated list of 2x5 `double`.
The full array `a` in all elements of `s`.	`s.a`	12-item comma-separated list of 2x5 `double`.
Selected elements of `a` in one element of `s`. Multiple elements of `s` are not allowed with this syntax.	`s(3,4).a(2,3:5)`	1x3 array of `double`.

The first three of these indexing expressions provide no access to individual elements of the field arrays. You could use these expressions to copy, rearrange, or delete parts of the structure.

Assigning Struct Values to a Comma-Separated List

Accessing a single value from a field in a struct array is no different from accessing one of the elements of any other MATLAB data type. You specify the appropriate subscripts for the struct and field arrays and MATLAB returns the value stored at that location in the array.

Accessing multiple elements, however, can be quite different. Multiple elements of a struct array cannot be assigned to a single variable because they do not necessarily belong to the same class. Instead, MATLAB assigns values from a struct array to a series of separate variables called a comma-separated list.

Create a struct `S`, with one field, `A`:

```
S = struct('A', {5, 'Anne', @myfun, 1:5});
```

Examining field `A` for all elements of `S` returns a comma-separated list:

```
S.A
ans =
     5
ans =
     Anne
ans =
     @myfun
ans =
     1     2     3     4     5
```

The potential problem with this type of output is that MATLAB overwrites the `ans` variable for each value returned. If you only want to display these values, then this command should suit your purpose. The next section shows how to assign to variables that you can reuse.

Assigning Struct Values to Separate Variables

If you were to assign multiple elements of a struct array to just one variable, MATLAB uses that variable to return the first value, but is unable to return all values of the array:

```
x = s.a
x =
     5
```

If you know how many values there are in the struct array elements you are trying to access, then you can provide that many outputs in the command, as shown here:

```
[v1 v2 v3 v4] = s.a

v1 =
     5
v2 =
     Anne
v3 =
     @myfun
v4 =
     1     2     3     4     5
```

As in the previous example, this is a comma-separated list. Each return variable is of the class and size of the struct array element assigned to it:

```
whos v1
  Name      Size             Bytes  Class     Attributes

  v1        1x1                  8  double

whos v2
  Name      Size             Bytes  Class    Attributes

  v2        1x4                  8  char
```

Assigning Struct Values to a Cell Array

You can assign the values of the struct array to a cell array using the following syntax:

```
[x{1:4}] = s.a
x =
    [5]    'Anne'    @myfun    [1x6 double]

whos x
  Name      Size            Bytes  Class    Attributes

  x         1x4               320  cell
```

Preallocating the cell array to a certain size and shape gives you control over how MATLAB returns the output:

```
x1 = cell(4,1);        %  Make x1 a 4-by-1 array
[x1{:}] = s.a
x1 =
    [         5]
    'Anne'
    @myfun
    [1x6 double]

x2 = cell(2,2);        %  Make x2 a 2-by-2 array
[x2{:}] = s.a
x2 =
    [   5]    @myfun
    'Anne'    [1x6 double]

x3 = cell(1,3);        %  Make x3 a 3-element array
[x3{:}] = s.a
x3 =
    [5]    'Anne'    @myfun
```

Another way to do this is to use the following statements:

```
[x1{1:4,1}] = s.a
[x2{1:2,1:2}] = s.a
[x3{1:3}] = s.a
```

Using Structures with Functions

This section describes how to apply a function to data contained within a structure field using the structfun function, and also how to pass arguments to and from a function using structures.

Applying a Function to the Fields of a Structure

Use the structfun function to execute a function on each field of a scalar struct array. This example executes an anonymous function on a structure having fields that name the days of a week. The anonymous function, @(x)x(1:3), shortens each string to its first three characters. The function reference page for structfun explains the use of the UniformOutput option:

```
days.f1 = 'Sunday';    days.f2 = 'Monday';
days.f3 = 'Tuesday';   days.f4 = 'Wednesday';
days.f5 = 'Thursday';  days.f6 = 'Friday';
days.f7 = 'Saturday';

shortNames = structfun(@(x)x(1:3), days, 'UniformOutput', false)
shortNames =
    f1: 'Sun'
    f2: 'Mon'
    f3: 'Tue'
    f4: 'Wed'
    f5: 'Thu'
    f6: 'Fri'
    f7: 'Sat'
```

See the reference page for structfun for additional help on using this function.

Passing Arguments in a Structure

A simple and easily maintainable way to pass arguments to or from a function is to package them in a structure, and then pass the entire structure to the function. This example passes information pertaining to four United States presidents to the function showPresInfo using only one input argument. You can also use struct arrays to return data from a function call.

Store data on the presidents in a 1-by-30 struct array:

```
USPres(27).name = 'William Howard Taft';   % 27th US President
USPres(27).term = [1909, 1913];            % Term
USPres(27).vp = 'James S. Sherman';        % Vice President

USPres(28).name = 'Woodrow Wilson';        % 28th
USPres(28).term = [1913, 1921];
USPres(28).vp = 'Thomas R. Marshall';

USPres(29).name = 'Warren G. Harding';     % 29th
USPres(29).term = [1921, 1923];
USPres(29).vp = 'Calvin Coolidge';

USPres(30).name = 'Calvin Coolidge';       % 30th
USPres(30).term = [1923, 1929];
USPres(30).vp = 'Charles Dawes';
```

Write a short program to display the information passed in:

```
function showPresInfo(number, info)
info(number-26)
```

Call this program, passing rows 27 through 30 of the struct array:

```
showPresInfo(29, USPres(27:30))
ans =
    name: 'Warren G. Harding'
    term: [1921 1923]
      vp: 'Calvin Coolidge'
```

Passing Selected Fields in a Structure

You can also pass selected fields of a structure in a function call. This example passes the `vp` field of the `USPres` struct array in the form of a comma-separated list. In this case, the function being called, `showVPInfo`, receives these strings as four separate input arguments:

The value passed to this function is a list of four separate items:

```
USPres(27:30).vp
ans =
    James S. Sherman
```

```
ans =
    Thomas R. Marshall
ans =
    Calvin Coolidge
ans =
    Charles Dawes
```

Write a short program that displays the name of a selected Vice President. Use the `varargin` function to accept and unpack the four separate input arguments generated by the `USPres(27:30).vp` input:

```
function showVPInfo(number, varargin)
str = ['The Vice President who served with ', ...
    'the %dth US President was %s\n'];
fprintf(str, number, varargin{number-26})
```

Run the program a couple of times to verify that the names of more than one Vice Presidents were passed:

```
showVPInfo(28, USPres(27:30).vp)
The Vice President who served with the 28th US President was Thomas R. Ma

showVPInfo(30, USPres(27:30).vp)
The Vice President who served with the 30th US President was Charles Dawe
```

Converting Between Struct Array and Cell Array

The `struct2cell` function converts a structure array to a cell array. The statement

```
c = struct2cell(s)
```

converts an `m`-by-`n` structure `s` that has `p` fields into a `p`-by-`m`-by-`n` cell array `c`:

The `cell2struct` function converts a cell array to a struct array. The statement

```
s = cell2struct(c,f,d)
```

converts a cell array `c` into a struct array `s` having the fields named in `f` and based on the `d` axis of the input cell array.

Conversion Example

This example converts a 1-by-2 struct array `USPres_s1` with four fields to a 4-by-1-by-2 cell array `USPres_c1`, and then back to a structure `USPres3_s2` that is equal to the original.

Create the original structure:

```
USPres_s1(1).name = 'Franklin D. Roosevelt';
USPres_s1(1).party = 'Democratic';
USPres_s1(1).term = {1933, 1945};
USPres_s1(1).vp = ...
    {'John Garner';'Henry Wallace';'Harry S Truman'};

USPres_s1(2).name = 'Harry S Truman';
USPres_s1(2).party = 'Democratic';
USPres_s1(2).term = {1945, 1953};
USPres_s1(2).vp = 'Alben Barkley';
whos USPres_s1
  Name           Size             Bytes  Class     Attributes

  USPres_s1      1x2               1400  struct
```

Convert the struct array to a cell array:

```
USPres_c1 = struct2cell(USPres_s1);
whos USPres_c1
  Name           Size              Bytes  Class    Attributes

  USPres_c1      4x1x2              1144  cell
```

Convert back to a struct array and compare it with the original:

```
USPres_s2 = cell2struct(USPres_c1, ...
    {'name','party','term','vp'}, 1);
whos USPres_s2
  Name           Size             Bytes  Class     Attributes

  USPres_s2      1x2               1400  struct

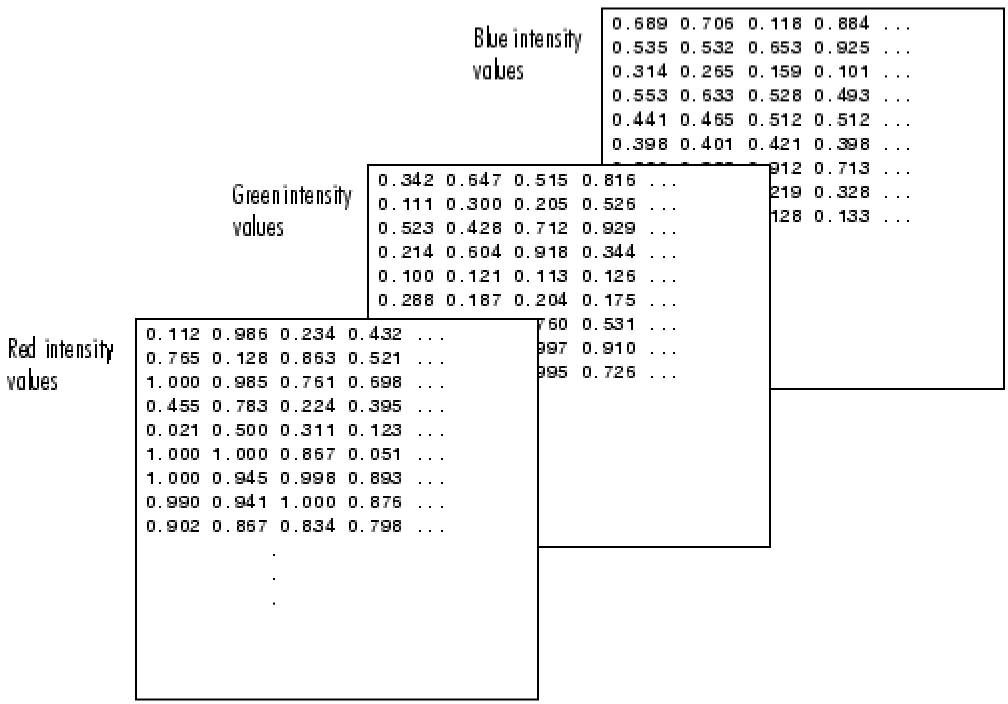
isequal(USPres_s1, USPres_s2)
ans =
```

1

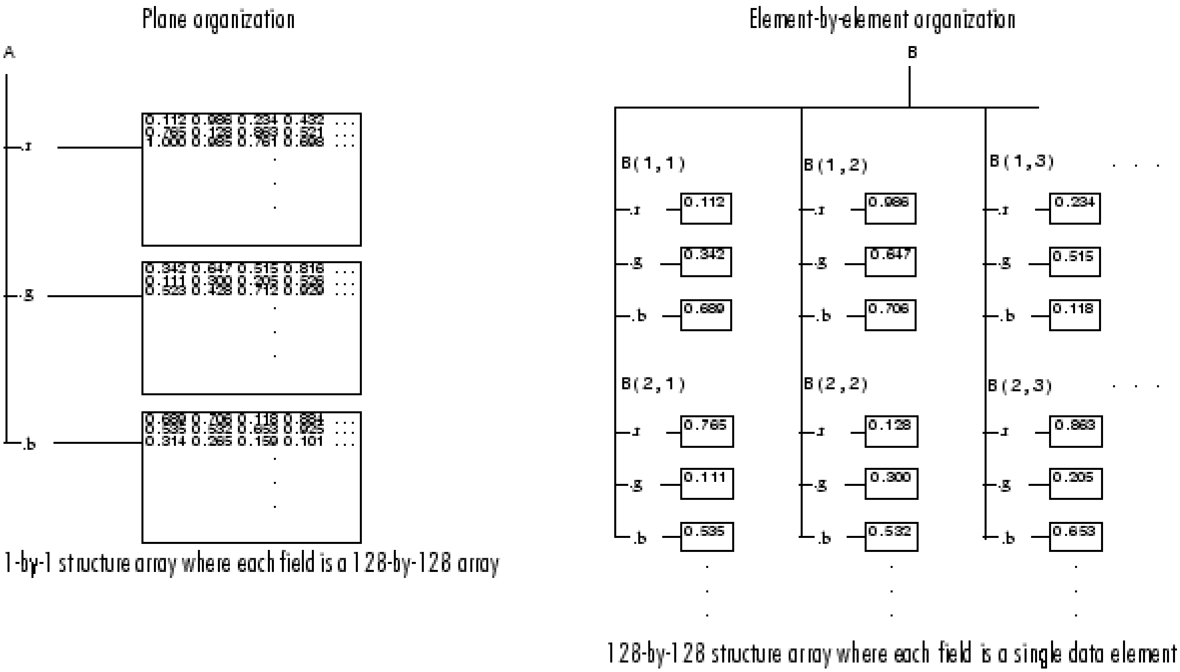
Organizing Data in Structure Arrays

The key to organizing structure arrays is to decide how you want to access subsets of the information. This, in turn, determines how you build the array that holds the structures, and how you break up the structure fields.

For example, consider a 128-by-128 RGB image stored in three separate arrays; `RED`, `GREEN`, and `BLUE`.



There are at least two ways you can organize such data into a structure array: plane organization and element-by-element organization.



Plane Organization

In the plane organization, shown to the left in the figure above, each field of the structure is an entire plane of the image. You can create this structure using

```
A.r = RED;
A.g = GREEN;
A.b = BLUE;
```

This approach allows you to easily extract entire image planes for display, filtering, or other tasks that work on the entire image at once. To access the entire red plane, for example, use

```
redPlane = A.r;
```

Plane organization has the additional advantage of being extensible to multiple images in this case. If you have a number of images, you can store them as `A(2)`, `A(3)`, and so on, each containing an entire image.

The disadvantage of plane organization is evident when you need to access subsets of the planes. To access a subimage, for example, you need to access each field separately:

```
redSub = A.r(2:12,13:30);
greenSub = A.g(2:12,13:30);
blueSub = A.b(2:12,13:30);
```

Element-by-Element Organization

The element-by-element organization, shown to the right in the previous figure, has the advantage of allowing easy access to subsets of data. However, it has the disadvantage of being very memory-intensive. To set up the data in this organization, use

```
for m = 1:size(RED,1)
   for n = 1:size(RED,2)
      B(m,n).r = RED(m,n);
      B(m,n).g = GREEN(m,n);
      B(m,n).b = BLUE(m,n);
   end
end
```

With element-by-element organization, you can access a subset of data with a single statement:

```
Bsub = B(1:10,1:10);
```

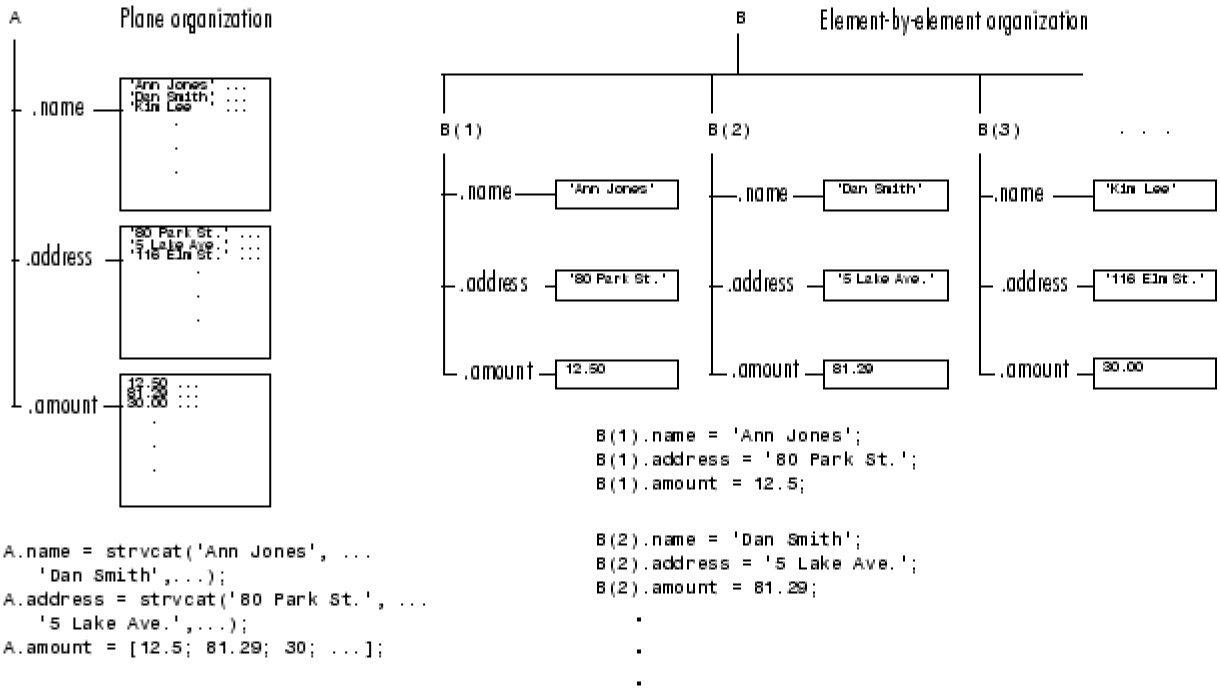
To access an entire plane of the image using the element-by-element method, however, requires a loop:

```
redPlane = zeros(128, 128);
for k = 1:(128 * 128)
    redPlane(k) = B(k).r;
end
```

Element-by-element organization is not the best structure array choice for most image processing applications. However, it can be the best for other applications wherein you routinely need to access corresponding subsets of structure fields. The example in the following section demonstrates this type of application.

Example — A Simple Database

Consider organizing a simple database.



Each of the possible organizations has advantages depending on how you want to access the data. Typically, your data does not dictate the organization scheme you choose. Rather, you must consider how you want to access and operate on the data.

Advantages of Using Plane Organization. Plane organization makes it easier to operate on all field values at once. If, for example, you want to find the average of all the values in the amount field,

- **Using plane organization**, you could use the following statement:

```
avg = mean(A.amount);
```

- **Using element-by-element organization**, you need to remember to enclose the amount expression in brackets:

```
avg = mean([B.amount]);
```

Advantages of Using Element-By-Element Organization. Element-by-element organization makes it easier to access all the information related to a single client. Consider an M-file, client.m, which displays the name and address of a given client on screen.

Using plane organization, the client function appears as:

```
function client(name,address)
disp(name)
disp(address)
```

Call this function as follows:

```
client(A.name(2,:), A.address(2,:))
```

Using element-by-element organization, both the function and function call require less code. The client function is:

```
function client(B)
disp(B)
```

To call the function, use:

```
client(B(2))
```

Element-by-element organization also makes it easier to expand the string array fields. If you do not know the maximum string length ahead of time for plane organization, you might need to frequently recreate the name or address field to accommodate longer strings.

Operator Summary

This section summarizes the following types of operators that work with structures:

- "Operators That Construct the Array" on page 1-97
- "Operators That Concatenate Structures" on page 1-97
- "Operators Used for Array Indexing " on page 1-97

Operators That Construct the Array

Syntax	Description
`S = struct('f1', x, 'f2', y, 'f3', z)`	Builds a 1-by-1 struct array `S` with fields `f1`, `f2`, and `f3`, which can contain data of unlike types. Field `f1` contains the value of `x`, field `f2` contains the value of `y`, etc.
`S = struct('f1', {x, y, z})`	Builds a 1-by-3 struct array `S` where each element of `S` has one field `f1`, which can contain data of unlike types. `S(1).f1` contains the value of `x`, `S(2).f1` contains the value of `y`, etc.

Operators That Concatenate Structures

Syntax	Description
`S3 = [S1 S2]`	Concatenates struct arrays `S1` and `S2` into a two-element struct array.
`S3 = [S1.F; S2.F]`	Concatenates the *fields* of struct arrays `S1` and `S2` into an array of the same data type. `S3` is not necessarily a struct.

Operators Used for Array Indexing

Syntax	Description
`X = S(s)`	Returns the elements of struct array `S` specified by subscripts `s`.
`X = S(s).F`	Returns all elements of field `F` for elements of `S` specified by `s`.

Syntax	Description
`X = S(s).F(f)`	Returns selected elements of field `F` for structure elements specified by subscript `s`.
X = S1(s_1).S2(s_2).F	Returns the contents of a nested struct array. Multiple elements of `S1` are not allowed with this syntax.
`X = S(s).F{c}`	Returns the specified elements of a cell array that reside in a field of struct array `S`.

Function Summary

This section summarizes the following types of functions that work with structures:

- "Functions Related to Constructing the Array" on page 1-98
- "Functions Related to the Type of the Array" on page 1-99
- "Functions Related to Struct Fields" on page 1-99
- "Functions Related to Applying Functions to a Struct Array" on page 1-99

Functions Related to Constructing the Array

Function	Description
`cat`	Concatenate arrays along specified dimension.
`horzcat`	Concatenate arrays horizontally.
`length`	Length of vector.
`ndims`	Number of array dimensions.
`numel`	Number of elements in array or subscripted array expression.
`repmat`	Replicate and tile array.
`reshape`	Reshape array.
`size`	Size of array.

Function	Description
struct	Create structure array.
vertcat	Concatenate arrays vertically.

Functions Related to the Type of the Array

Function	Description
cell2struct	Convert cell array to structure array.
class	Create object or return class of object.
isstruct	Determine whether input is structure array.
struct2cell	Convert structure to cell array.
whos	List variables in workspace.

Functions Related to Struct Fields

Function	Description
fieldnames	Get all field names of specified structure.
getfield	Get contents of the specified field.
isempty	Determine whether array is empty.
isfield	Determine if input is a structure field.
orderfields	Order fields of a structure array.
rmfield	Remove structure field.
setfield	Set structure field contents, returning the modified structure.
Dynamic Fieldnames	Generate field name strings at run time.

Functions Related to Applying Functions to a Struct Array

Function	Description
structfun	Apply function to each field of scalar structure.

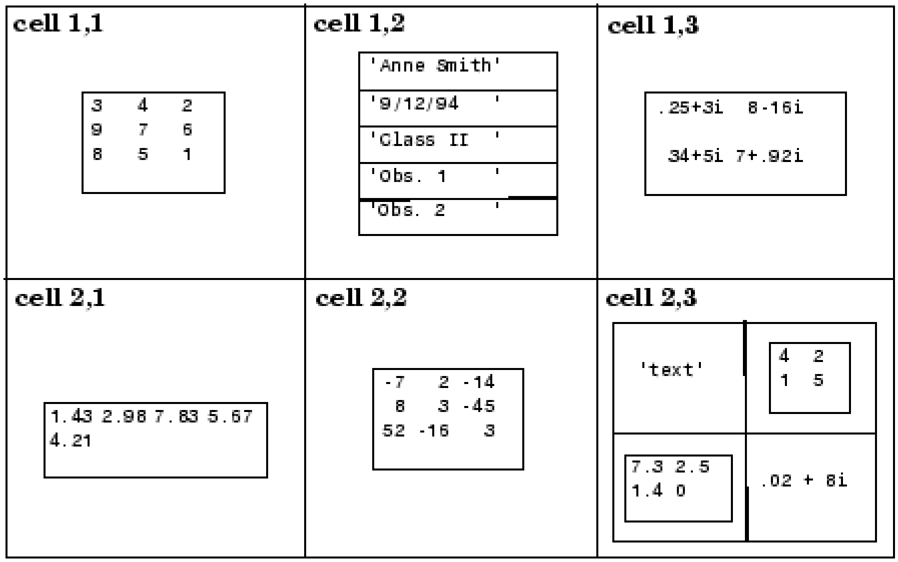
Function	Description
`varargin`	Variable length input argument list.
`varargout`	Variable length output argument list.

Cell Arrays

In this section...

What Is a Cell Array?

A cell array is a collection of containers called *cells* in which you can store different types of data. The figure shown below represents a 2-by-3 cell array. The cells in row one hold an array of unsigned integers, an array of strings, and an array of complex numbers. Row two holds three other types of arrays, the last being a second cell array nested in the outer one:



Each cell of a cell array contains some type of MATLAB array. The data in this array can belong to any one MATLAB or user-defined class, and can have any valid array dimensions; this includes 1-by-1 (a scalar array), or having one or more dimension equal to zero (an empty array). A second cell of the same array can belong to an entirely different class, and can also have different dimensions than the first. The capability to store arrays of mixed classes and sizes is the most significant feature of a cell array. Another common use of cell arrays is to store character strings that are of unequal length. A cell array that is used for this purpose is called a *cell array of strings*.

Like all MATLAB arrays, cell arrays must be rectangular in shape. That is, the length of all rows must be the same, the length of all columns the same, and so on for every dimension of the array.

In many respects, cell arrays are quite similar to struct arrays. See "Comparing Struct Arrays with Cell Arrays" on page 1-72 for help in deciding which of these two classes best suits the needs of your applications.

Cell Array Operations

This table shows the operators used in creating, concatenating, and indexing into the cells of a cell array.

Operation	Syntax	Description
Creating	`C = {A B D E}`	Builds a cell array `C` that can contain data of unlike types in `A`, `B`, `D`, and `E`.
Concatenating	`C3 = {C1 C2}`	Concatenates cell arrays `C1` and `C2` into a two-element cell array `C3` such that `C3{1} = C1` and `C3{2} = C2`.
	`C3 = [C1 C2]`	Concatenates the *contents* of cell arrays `C1` and `C2`, assuming that the dimensions of these arrays are compatible.
Indexing	`X = C(s)`	Returns the *cells* of array `C` that are specified by subscripts `s`.
	`X = C{s}`	Returns the *contents* of the cells of `C` that are specified by subscripts `s`.
	`X = C{s}(t)`	References one or more elements of an array that resides within a cell. Subscript `s` selects the cell, and subscript `t` selects the array element(s).

For more information on these operations, see “Creating a Cell Array” on page 1-103, “Concatenating Cell Arrays” on page 1-108, and “Indexing into a Cell Array” on page 1-109, respectively.

Creating a Cell Array

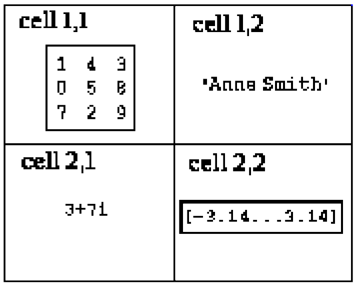
Creating cell arrays in MATLAB is similar to creating arrays of other MATLAB classes like `double`, `char`, and so on. The main difference is that, when creating a cell array, you enclose the array contents or indices with curly braces `{ }` instead of square brackets `[ ]`. The curly braces are cell array

constructors, just as square brackets are numeric array constructors. Use commas or spaces to separate elements and semicolons to terminate each row.

For example, to create a 2-by-2 cell array `A`, type

```
A = {[1 4 3; 0 5 8; 7 2 9], 'Anne Smith'; 3+7i, -pi:pi/4:pi};
```

This results in the array shown below.



Note The curly braces operator creates two-dimensional matrices only, (including 0-by-0, 1-by-1, and 1-by-n matrices). To create cell arrays of more than two dimensions, see "Creating Multidimensional Arrays".

Nesting One Cell Array in Another

To nest one cell array within another, enclose both inner and outer cell arrays with the curly braces `{ }`. The example shown here nests a cell array of vital signs inside a cell array of a person's medical record. (Defining the columns with a `header` is not usually required, and is just used here to make the example simpler):

```
header = {'Name', 'Age', 'Pulse/Temp/BP'};
records(1,:) = {'Kelly', 49, {58, 98.3, [103, 72]}};

header, records
header =
    'Name'    'Age'    'Pulse/Temp/BP'
records =
    'Kelly'    [49]    {1x3 cell}
```

It is often easier to build a nested cell array in steps. The example below creates the inner cell array, `vitalsigns`, first. The second statement then uses the `vitalsigns` array in creating the outer cell array, `records`:

```
vitalsigns = {60, 98.4, [105, 75]};

records(1,:) = {'Kelly', 49, vitalsigns}
record =
    'Name'     'Age'    'Pulse/Temp/BP'
    'Kelly'    [ 49]         {1x3 cell}
```

Verify the new values in the `records` cell array::

```
fprintf('pulse: %d   temp: %3.1f   bp: %d/%d\n', ...
        records{3}{:})
pulse: 60   temp: 98.4   bp: 105/75
```

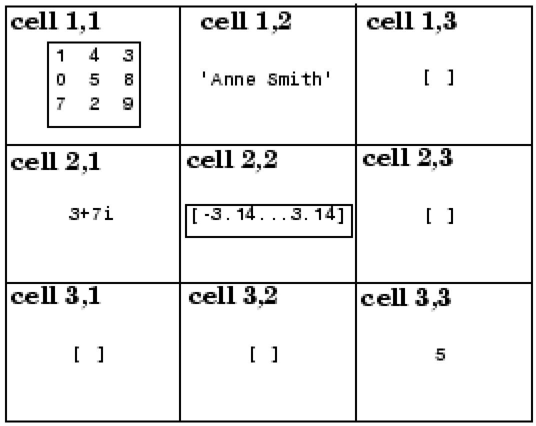
Creating Cell Arrays One Cell At a Time

You also can create a cell array one cell at a time by using multiple assignment statements. MATLAB expands the size of the cell array with each assignment statement:

```
A(1,1) = {[1 4 3; 0 5 8; 7 2 9]};
A(1,2) = {'Anne Smith'};
A(2,1) = {3+7i};
A(2,2) = {-pi:pi/4:pi};
```

If you assign data to a cell that is outside the dimensions of the current array, MATLAB automatically expands the array to include the subscripts you specify. It fills any intervening cells with empty matrices. For example, the assignment below turns the 2-by-2 cell array `A` into a 3-by-3 cell array:

```
A(3,3) = {5};
```



3–by-3 Cell Array

Note If you already have a numeric array of a given name, don't try to create a cell array of the same name by assignment without first clearing the numeric array. If you do not clear the numeric array, MATLAB assumes that you are trying to *mix* cell and numeric syntaxes, and generates an error. Similarly, MATLAB does not clear a cell array when you make a single assignment to it. If any of the examples in this section give unexpected results, clear the cell array from the workspace and try again.

Handling Unassigned Cells. To keep all dimensions of a cell array even, MATLAB automatically fills in any unassigned cells as you build the cell array. For example, if you have a cell array that consists of a row of three elements, and you add one new cell to a second row, MATLAB adds two cells to the new row to keep all rows at the same length. The values of these two cells are set to the empty array `[]`. This is called *scalar expansion*.

MATLAB handles other array types in a similar manner, except that it sets unassigned elements to zero instead of the empty array. This example adds a single element to a 1-by-3 array of type `double`, and then does the same to a cell array:

```
A = [2 4 6];   A(2,1) = 8
A =
      2      4      6
```

```
     8     0     0

C = {2 4 6};   C(2,1) = {8}
C =
    [2]    [4]    [6]
    [8]     []     []
```

Alternative Assignment Syntax

When assigning values to a cell array, either of the syntaxes shown below is valid. You can use the braces on the right side of the equation, enclosing the value being assigned, as shown here:

```
A(1,1) = {[1 4 3; 0 5 8; 7 2 9]};
A(1,2) = {'Anne Smith'};
```

You can also use them on the left side, enclosing the array subscripts:

```
A{1,1} = [1 4 3; 0 5 8; 7 2 9];
A{1,2} = 'Anne Smith';
```

Preallocating Memory for the Array

MATLAB stores internal information for a cell array in a contiguous segment of memory called a *header*. If you increase the number of cells in a cell array over time, the size of the header also grows, thus using more of this segment in memory. This can eventually lead to “out of memory” errors.

If you can roughly estimate the dimensions of a cell array at the time you create it, you can preallocate the necessary space in memory and help to avoid this type of problem. See the documentation on Data Structures and Memory to help you make this estimate.

Note Unlike the internal header information of a cell array, the memory consumed by the *data* stored in a cell array is not contiguous. While preallocating memory can help avoid memory problems when increasing the dimensions of a cell array, it does not protect against shortages in memory due to the amount of data you store in the array. Even with preallocated cell (and struct) arrays, you need to take precautions against using more memory than is available.

How to Preallocate Memory. To allocate memory for a 25-by-50 cell array and initialize the entire array to `[]`, use either of the following two methods:

```
C = cell(25,50);
C{25,50} = [];
```

After the memory has been allocated, you can begin to construct the array by assigning data to it.

Concatenating Cell Arrays

There are two ways that you can create a new cell array from existing cell arrays:

- Concatenate entire cell arrays to individual cells of the new array. For example, join three cell arrays together to build a new cell array having three elements, each containing a cell array. This method uses the curly brace `{ }` operator.
- Concatenate the *contents* of the cells into a new array. For example, join cell arrays of size `m-by-n1`, `m-by-n2`, and `m-by-n3` together to yield a new cell array that is `m-by-(n1+n2+n3)` in size. This method uses the square bracket `[ ]` operator.

Here is an example. First, create three 3-row cell arrays of different widths:

```
C1 = {'Jan' 'Feb';  '10' '17';  uint16(2004) uint16(2001)};
C2 = {'Mar' 'Apr' 'May';  '31' '2' '10';  ...
      uint16(2006) uint16(2005) uint16(1994)};
C3 = {'Jun';  '23';  uint16(2002)};
```

This creates arrays C1, C2, and C3:

```
      C1                      C2                    C3
'Jan'    'Feb'      'Mar'    'Apr'    'May'       'Jun'
'10'     '17'       '31'     '2'      '10'        '23'
[2004]   [2001]     [2006]   [2005]   [1994]      [2002]
```

Use the curly brace operator to concatenate entire cell arrays, thus building a 1-by-3 cell array from the three initial arrays. Each cell of this new array holds its own cell array:

```
C4 = {C1 C2 C3}
C4 =
    {3x2 cell}    {3x3 cell}    {3x1 cell}
```

Now use the square bracket operator on the same combination of cell arrays. This time MATLAB concatenates the contents of the cells together and produces a 3-by-6 cell array:

```
C5 = [C1 C2 C3]
C5 =
    'Jan'     'Feb'     'Mar'     'Apr'     'May'     'Jun'
    '10'      '17'      '31'      '2'       '10'      '23'
    [2004]    [2001]    [2006]    [2005]    [1994]    [2002]
```

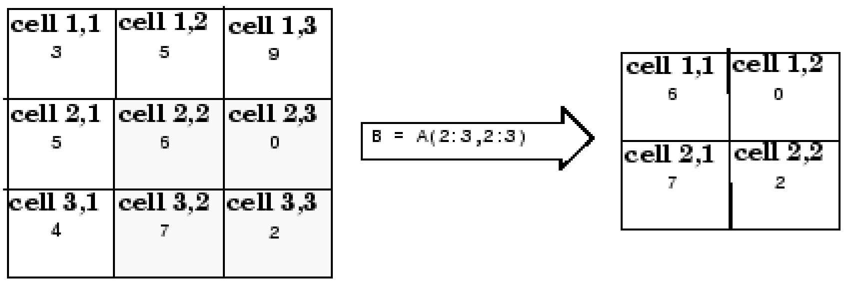
Note The notation {} denotes the empty cell array, just as [] denotes the empty matrix for numeric arrays. You can use the empty cell array in any cell array assignments.

Indexing into a Cell Array

When working with cell arrays, you have a choice of selecting entire cells of an array to work with, or the contents of those cells. The first method is *cell indexing*, the second is *content indexing*:

- Cell indexing enables you to work with whole cells of an array. You can access single or multiple cells within the array, but you cannot select anything less than the complete cell. If you want to manipulate the cells of an array without regard to the contents of those cells, use cell indexing. This type of indexing is denoted by the parentheses operator `( )`.

Use cell indexing to assign any set of cells to another variable, creating a new cell array.



Creating a New Cell Array from an Existing One

- Content indexing gives you access to the contents of a cell. You can work with individual elements of an array within a cell, but you can only do so for one cell at a time. This indexing uses the curly brace operator `{ }`.

Note The examples in this section focus on two-dimensional cell arrays. For examples of higher-dimension cell arrays, see "Multidimensional Arrays".

This example shows how to use cell and content indexing. Start out by creating the following 3-by-3 cell array. The third element of each row is a nested cell array:

```
header = {'Name', 'Age', 'Pulse/Temp/BP'};
records(1,:) = {'Kelly', 49, {58, 98.3, [103, 72]}};
records(2,:) = {'Mark', 25,  {60, 98.6, [105, 75]}};
records(3,:) = {'Susan', 32, {71, 99.1, [110, 78]}};
```

Display the contents of the cell array. Defining the columns with a `header` is not usually required, and is just used here to make the example simpler:

```
header =
    'Name'    'Age'    'Pulse/Temp/BP'
records =
    'Kelly'    [49]    {1x3 cell}
    'Mark'     [25]    {1x3 cell}
    'Susan'    [32]    {1x3 cell}
```

Use content indexing (curly braces) to change one of the names. Content indexing gives you access to what is contained within the cells of the array:

```
records{3,1}='Susanne'
records =
    'Kelly'      [49]    {1x3 cell}
    'Mark'       [25]    {1x3 cell}
    'Susanne'    [32]    {1x3 cell}
```

Use cell indexing (parentheses) to delete an entire row. (You delete part of a cell array by assigning the empty array `[]` to it.) Cell indexing is appropriate here because you do not need access to the *contents* of the row:

```
records(1,:) = []
records =
    'Mark'       [25]    {1x3 cell}
    'Susanne'    [32]    {1x3 cell}
```

Indexing Into Inner Levels of the Cell Array

The cells of a cell array contain arrays of standard MATLAB data types. These arrays use the indexing syntax appropriate to the class of the array. The table below shows examples of statements that use a combination of cell and struct array indexing:

Action	Required Indexing
Access an element of an array in a cell of cell array `C`.	`C{3,15}(5,25)`
Access an element of array `A`, where `A` is a field of a structure that resides in cell array `C`.	`C{3,15}.A(5,20)`
Access an element of an array that resides in a nested cell array.	`C{3,15}{5,20}(50,5)`
Access an element of array `B`, where `B` is a field of structure `A`, and `A` is a field of a structure that resides in cell array `C`.	`C{3,15}.A(5,20).B(50,5)`
Access an element of cell array `B`, where `B` is a field of a structure that resides in cell array `C`.	`C{3,15}.B{5,20}`

Start this example by creating the `records` cell array taken from the example in the previous section:

```
records(1,:) = {'Kelly', 49, {58, 98.3, [103, 72]}};
records(2,:) = {'Mark', 25,  {60, 98.6, [105, 75]}};
records(3,:) = {'Susan', 32, {71, 99.1, [110, 78]}};
```

Display information from cells in the nested cell array. This requires two adjacent expressions of content indexing, `{2,3}{3}`:

```
fprintf('Name: %s   Systolic: %d   Diastolic: %d\n', ...
        records{2,1}, records{2,3}{3})
Name: Mark   Systolic: 105   Diastolic: 75
```

Indexing Tips

You can index into a nested array in stages rather than all at once. Consider breaking down this indexing expression

```
C{5,3}{4,7}(:,4)
```

into the following:

```
x = C{5,3};      % x is a cell array
y = x{4,7};      % y is also a cell array
z = y(:,4)       % z is a standard array
```

See the section on "Indexing Tips" on page 1-83 in the documentation on "Structures" for indexing tips that apply to both the `cell` and `struct` classes.

Using Map Objects in Cell Array Indexing

If you want both the numeric indexing of cell arrays and the named containers of structures, you can combine the two to some extent by implementing cell array indexing with a MATLAB Map object (see "Map Containers" on page 1-144). The Map object provides a translation from a name string to a numeric array index. This implementation has the advantage of using less memory than a struct or cell array, but has the disadvantage of being slower.

The example shown here demonstrates the use of a Map object in locating information in a cell array. Note that the names given to the keys of a Map

object do not have to adhere to the rules for variable names. In this example, each of the key names contain a space character. This is not allowed in variable names:

```
redSoxStats(57:60,1:4) = { ...
%
%   AtBat  Runs  Hits  HR
%
     653,  118,  213,  17;  ... % Pedroia
     554,   98,  155,   9;  ... % Ellsbury
     538,   91,  168,  29;  ... % Youkilis
     423,   37,   93,  13}; ... % Varitek

m1 = containers.Map({'Dustin Pedroia','Jacoby Ellsbury', ...
     'Kevin Youkilis', 'Jason Varitek'}, {57,58,59,60});

m2 = containers.Map({'AtBat','Runs','Hits', 'HR'}, ...
   {1,2,3,4});

player = 'Dustin Pedroia';
fprintf( ...
   '\n   %s had %d At Bats and %d hits this season.\n', ...
   player, redSoxStats{m1(player), m2('AtBat')}, ...
   redSoxStats{m1(player), m2('Hits')})

Dustin Pedroia had 653 At Bats and 213 hits in the 2008 season.
```

Assigning Values to a Cell Array

Use the curly brace `{ }` operator on the right side of the statement to assign values to a cell array:

To store four values in a 2-by-2 cell array, use

```
C = {magic(5), 'Hello'; uint8(100), [1:3:19]}
C =
    [5x5 double]    'Hello'
    [       100]    [1x7 double]
```

The following commands place the values into different cells of cell array `C`:

```
clear C
C(3,1:4) = {magic(5), 'Hello', uint8(100), [1:3:19]}
C =
             []          []        []               []
             []          []        []               []
    [5x5 double]    'Hello'    [100]    [1x7 double]
```

```
clear C
C(2:3,5:6) = {magic(5), 'Hello'; uint8(100), [1:3:19]}
C =
     []     []     []     []               []               []
     []     []     []     []    [5x5 double]    [        100]
     []     []     []     []    'Hello'         [1x7 double]
```

The `deal` function offers an alternative method of writing to the cell array. These two statements produce the same result as the statements shown above that use the curly braces `{ }` operator:

```
[C{3,1:4}] = deal(magic(5), 'Hello', uint8(100), [1:3:19]);
[C{2:3,5:6}] = deal(magic(5), 'Hello', uint8(100), [1:3:19]);
```

Returning Data from a Cell Array

This section describes the syntax to use to have MATLAB return data from a cell array, how to assign data from a cell array to a comma-separated list or separate output variables, and also how to plot the contents of a cell array.

Obtaining Values from the Array

The following table shows a number of different ways of returning data from a cell array. The variable `c` is a 3x4 cell array in which each cell contains a 2x5 array of class `double`.

Values to be acquired	MATLAB Statement	Data Structure Returned
Top level of cell array `c`	`c`	3x4 cell array.
Top level of cell array `c`, as a vector	`c(:)`	12x1 cell array.
Selected cells in cell array `c`	`c(2:3,1:3)`	2x3 cell array.

Values to be acquired	MATLAB Statement	Data Structure Returned
Full contents of one cell in cell array `c`.	`c{2,3}`	2x5 array of `double`.
Full contents of selected cells in cell array `c`.	`c{2:3,3:4}`	4-item comma-separated list of 2x5 `double`.
Full contents of all cells in cell array `c`.	`c{:}`	12-item comma-separated list of 2x5 `double`.
Selected elements of one cell in cell array `c`. You cannot use multiple elements of `c` with this syntax.	`c{3,4}(2,3:5)`	1x3 array of `double`.

The first three of these indexing expressions provide no access to individual elements of the cells. You could use these expressions to copy, rearrange, or delete parts of the cell array.

Assigning Cell Values to a Comma-Separated List

Accessing a single value from one cell of a cell array is no different from accessing one of the elements of any other MATLAB data type. Accessing multiple elements, however, can be quite different. Multiple elements of a cell array cannot be assigned to a single variable because they do not necessarily belong to the same class. Instead, MATLAB assigns values from a cell array to a series of separate variables called a comma-separated list. Here is an example of such a list:

First, create a 3-by-3 cell array called `records`:

```
records(1,:) = {'Kelly', 49, {58, 98.3, [103, 72]}};
records(2,:) = {'Mark', 25,  {60, 98.6, [105, 75]}};
records(3,:) = {'Susan', 32, {71, 99.1, [110, 78]}};
```

Displaying one column of the cell array causes MATLAB to return three separate values, each, in succession, assigned to the `ans` variable:

```
records{:,2}
ans =
    49
ans =
```

```
    25
ans =
    32
```

The potential problem with this type of output is that MATLAB overwrittes the ans variable for each value returned. If you only want to display these values, then this command should suit your purpose. The next section shows how to assign to variables that you can reuse.

Assigning Cell Values to Separate Variables

If you were to assign multiple elements of a cell array to just one variable, MATLAB uses that variable to return the first value, but is unable to return all values of the array:

```
x = records{:,2}
x =
    49
```

If you know how many values there are in the cell array elements you are trying to access, then you can provide that many outputs in the command, as shown here:

```
[v1 v2 v3] = records{:,1}
v1 =
    Kelly
v2 =
    Mark
v3 =
    Susan
```

As in the previous example, this is a comma-separated list. As you can see here, each return variable adopts the class and size of the cell array element assigned to it:

```
whos v1
  Name      Size            Bytes  Class    Attributes

  v1        1x5                10  char

whos v2
```

```
  Name      Size              Bytes  Class     Attributes

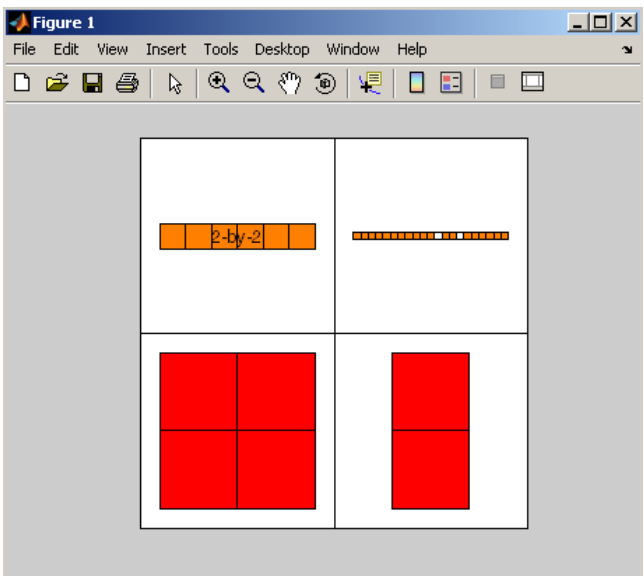
  v2        1x4                   8  char
```

Plotting the Cell Array

For a high-level graphical display of cell architecture, use the `cellplot` function. Consider a 2-by-2 cell array containing two text strings, a matrix, and a vector:

```
c{1,1} = '2-by-2';
c{1,2} = 'eigenvalues of eye(2)';
c{2,1} = eye(2);
c{2,2} = eig(eye(2));
```

The command `cellplot(c)` produces this figure.



Using Cell Arrays with Functions

This section describes how to apply a function to data contained within a cell array using the `cellfun` function, and also how to pass arguments to and from a function using cell arrays.

Applying a Function to the Cells of a Cell Array

Use the `cellfun` function to run a function on each field of a scalar cell array. This example runs an anonymous function on a cell array containing the days of a week. The anonymous function, `@(x)x(1:3)`, shortens each string to its first three characters. The function reference page for `cellfun` explains the use of the `UniformOutput` option:

```
days{1} = 'Sunday';    days{2} = 'Monday';
days{3} = 'Tuesday';   days{4} = 'Wednesday';
days{5} = 'Thursday';  days{6} = 'Friday';
days{7} = 'Saturday';

shortNames = cellfun(@(x)x(1:3), days, 'UniformOutput', false)
shortNames =
    'Sun'    'Mon'    'Tue'    'Wed'    'Thu'    'Fri'    'Sat'
```

See the reference page for `cellfun` for additional help on using this function.

Passing Variable Numbers of Arguments

You can call a function with variable numbers of input or output arguments by using the terms `varargin` and `varargout` in the respective input and output argument lists for that function. The function being called provides access to these arguments using cell arrays named `varargin` and `varargout`. See Passing Variable Numbers of Arguments in the documentation on "Functions and Scripts".

Passing Arguments in a Cell Array

A simple and easily maintainable way to pass arguments to or from a function is to package them in a cell array, and then pass the entire cell array to the function. This example passes information pertaining to four United States presidents to the function `showPresInfo`:

```
USPres = cell(30,3);    % Allocate memory for the array.
```

```
USPres{27,1} = 'William Howard Taft';   % 27th US President
USPres{27,2} = [1909, 1913];            % Term
USPres{27,3} = 'James S. Sherman';      % Vice President

USPres{28,1} = 'Woodrow Wilson';        % 28th
USPres{28,2} = [1913, 1921];
USPres{28,3} = 'Thomas R. Marshall';

USPres{29,1} = 'Warren G. Harding';     % 29th
USPres{29,2} = [1921, 1923];
USPres{29,3} = 'Calvin Coolidge';

USPres{30,1} = 'Calvin Coolidge';       % 30th
USPres{30,2} = [1923, 1929];
USPres{30,3} = 'Charles Dawes';
```

Write a short program to display the information passed in:

```
function showPresInfo(number, info)
info(number-26, :)'
```

Call this program, passing rows 27 through 30 of the cell array:

```
showPresInfo(29, USPres(27:30, :))
ans =
    'Warren G. Harding'
    [1x2 double]
    'Calvin Coolidge'
```

Passing Selected Cells of a Cell Array

You can also pass selected cells of a cell array in a function call. This example passes the names of the four Vice Presidents in the form of a comma-separated list. In this case, the function being called, `showVPInfo`, receives these strings as four separate input arguments:

The value passed to this function is a list of four separate items:

```
USPres(27:30).vp
ans =
```

```
    James S. Sherman
ans =
    Thomas R. Marshall
ans =
    Calvin Coolidge
ans =
    Charles Dawes
```

Write a short program that displays the name of a selected Vice President. Use the `varargin` function to accept and unpack the four separate input arguments generated by the `USPres{27:30,3}` input:

```
function showVPInfo(number, varargin)
str = ['The Vice President who served with ', ...
    'the %dth US President was %s\n'];
fprintf(str, number, varargin{number-26})
```

Run the program a couple of times to verify that the names of more than one Vice Presidents were passed:

```
showVPInfo(28, USPres{27:30,3})
The Vice President who served with the 28th US President was Thomas R. Ma

showVPInfo(30, USPres{27:30,3})
The Vice President who served with the 30th US President was Charles Dawe
```

Converting Between Cell Array and Struct Array

The `cell2struct` function converts a cell array to a struct array. The statement

```
s = cell2struct(c,f,d)
```

converts a cell array `c` into a struct array `s` having the fields named in `f` and based on the `d` axis of the input cell array.

The `struct2cell` function converts a structure array to a cell array. The statement

```
c = struct2cell(s)
```

converts an `m`-by-`n` structure `s` that has `p` fields into a `p`-by-`m`-by-`n` cell array `c`:

Conversion Example

This example converts a 4-by-1-by-2 cell array `USPres_c1` to a 1-by-2 struct array `USPres_s1` with four fields, and then back to a cell array `USPres_c2` that is equal to the original.

Create the original cell array:

```
USPres_c1{1,1,1} = 'Franklin D. Roosevelt';
USPres_c1{2,1,1} = 'Democratic';
USPres_c1{3,1,1} = [1933, 1945];
USPres_c1{4,1,1} = ...
     {'John Garner';'Henry Wallace';'Harry S Truman'};

USPres_c1{1,1,2} = 'Harry S Truman';
USPres_c1{2,1,2} = 'Democratic';
USPres_c1{3,1,2} = [1945, 1953];
USPres_c1{4,1,2} = {'Alben Barkley'};
whos USPres_c1
  Name           Size             Bytes  Class    Attributes

  USPres_c1      4x1x2              964  cell
```

Convert the cell array to a struct array:

```
USPres_s1 = cell2struct(USPres_c1, ...
     {'name','party','term','vp'}, 1);
whos USPres_s1
  Name           Size            Bytes  Class     Attributes

  USPres_s1      1x2              1220  struct
```

Convert back to a cell array and compare it with the original:

```
USPres_c2 = struct2cell(USPres_s1);
whos USPres_c2
  Name           Size             Bytes  Class    Attributes

  USPres_c2      4x1x2              964  cell

isequal(USPres_c1, USPres_c2)
ans =
```

1

Operator Summary

This section summarizes the following types of operators that work with cell arrays:

- "Operators That Construct the Cell Array" on page 1-122
- "Operators That Concatenate Cells and Cell Content" on page 1-122
- "Operators Used for Cell Array Indexing " on page 1-122

Operators That Construct the Cell Array

Syntax	Description
`C = {A B D E}`	Builds a cell array `C` that can contain data of unlike types in `A`, `B`, `D`, and `E`.

Operators That Concatenate Cells and Cell Content

Syntax	Description
`C3 = {C1 C2}`	Concatenates cell arrays `C1` and `C2` into a two-element cell array `C3`, such that `C3{1}` = `C1` and `C3{2}` = `C2`.
`C3 = [C1 C2]`	Concatenates the *contents* of cell arrays `C1` and `C2` into a new cell array with `length(C3) == length(C1) + length(C2)`.

Operators Used for Cell Array Indexing

Syntax	Description
`X = C(s)`	Returns the *cells* of array `C` that are specified by subscripts `s`.
`X = C{s}`	Returns the *contents* of the cells of `C` that are specified by subscripts `s`.

Syntax	Description
X = C{s}(v)	References one or more elements of an array that resides within a cell. Subscript s selects the single cell, and subscript v selects the array element(s).
X = C{s_1}{s_2}	Returns the contents of a nested cell array. Subscripts for the outer array C are s_1. These subscripts can only refer to one cell of the outer array. Subscripts for the inner cell array are s_2.
X = C{s_1}{s_2}(v)	Returns one or more elements of an array that reside in a nested cell array.
X = C{s}(t).f(v)	Returns one or more elements of an array that reside in a struct field, where the struct resides in a cell of cell array C. Subscripts are s for the cell array, t for the struct array, and v for the lowest-level array.

Function Summary

This section summarizes the following types functions that work with cell arrays:

- "Functions Related to Constructing the Array" on page 1-123
- "Functions Related to the Type of the Array" on page 1-124
- "Functions Related to Obtaining Cell Array Contents" on page 1-124
- "Functions Related to Applying Functions to a Cell Array" on page 1-124
- "Functions Used with Cell Array Conversion" on page 1-125

Functions Related to Constructing the Array

Function	Description
cat	Concatenate arrays along specified dimension.
cell	Create cell array.
horzcat	Concatenate arrays horizontally.

Function	Description
length	Length of array.
ndims	Number of array dimensions.
numel	Number of elements in array or subscripted array expression.
repmat	Replicate and tile array.
reshape	Reshape array.
size	Size of array.
vertcat	Concatenate arrays vertically.

Functions Related to the Type of the Array

Function	Description
cell2struct	Convert cell array to structure array.
class	Create object or return class of object.
iscell	Determine whether input is cell array.
struct2cell	Convert structure to cell array.
whos	List variables in workspace.

Functions Related to Obtaining Cell Array Contents

Function	Description
celldisp	Display cell array contents.
cellplot	Display a graphical depiction of a cell array.
deal	Copy input to separate outputs.

Functions Related to Applying Functions to a Cell Array

Function	Description
cellfun	Apply function to each field of scalar cell array.

Function	Description
varargin	Variable length input argument list.
varargout	Variable length output argument list.

Functions Used with Cell Array Conversion

Function	Description
cell2struct	Convert cell array into struct array.
struct2cell	Convert struct array into cell array.
mat2cell	Divide matrix into cell array of matrices
cell2mat	Convert cell array of matrices to single matrix
num2cell	Convert numeric array to cell array

Function Handles

In this section...

Overview

A *function handle* is a callable association to a MATLAB function. It contains an association to that function that enables you to invoke the function regardless of where you call it from. This means that, even if you are outside the normal scope of a function, you can still call it if you use its handle.

With function handles, you can:

- Pass a function to another function
- Capture data values for later use by a function
- Call functions outside of their normal scope
- Save the handle in a MAT-file to be used in a later MATLAB session

See "Applications of Function Handles" on page 1-131 for an explanation of each of these applications.

Creating a Function Handle

- "Handles to Anonymous Functions" on page 1-128
- "Function Handle Arrays" on page 1-129

You construct a handle for a specific function by preceding the function name with an @ sign. The syntax is:

```
h = @functionname
```

where h is the variable to which the returned function handle is assigned.

Use only the function *name*, with no path information, after the @ sign. If there is more than one function with this name, MATLAB associates with the handle the one function source it would dispatch to if you were actually calling the function.

Create a handle h for a function plot that is on your MATLAB path:

```
h = @plot;
```

Once you create a handle for a function, you can invoke the function by means of the handle instead of using the function name. Because the handle contains the absolute path to its function, you can invoke the function from any location that MATLAB is able to reach, as long as the M-file for the function still exists at this location. This means that functions in one M-file can call functions that are not on the MATLAB path, subfunctions in a separate M-file, or even functions that are private to another directory, and thus not normally accessible to that caller.

Maximum Length of a Function Name

Function names used in handles are unique up to N characters, where N is the number returned by the function namelengthmax. If the function name exceeds that length, MATLAB truncates the latter part of the name.

For function handles created for Sun™ Java™ constructors, the length of any segment of the package name or class name must not exceed namelengthmax characters. (The term *segment* refers to any portion of the name that lies before, between, or after a dot. For example, java.lang.String has three segments). The overall length of the string specifying the package and class has no limit.

Associating a Handle with a Function

At the time you create a function handle, MATLAB must decide exactly which function it is to associate the handle to. In doing so, MATLAB uses the same rules used to determine which M-file to invoke when you make a function call. To make this determination, MATLAB considers the following:

- Scope — The function named must be on the MATLAB path at the time the handle is constructed.
- Precedence — MATLAB selects which function(s) to associate the handle with, according to the function precedence rules described under Determining Which Function Gets Called.
- Overloading — If additional M-files on the path overload the function for any of the standard MATLAB classes, such as `double` or `char`, then MATLAB associates the handle with these M-files, as well.

M-files that overload a function for classes other than the standard MATLAB classes are not associated with the function handle at the time it is constructed. Function handles do operate on these types of overloaded functions, but MATLAB determines which implementation to call at the time of evaluation in this case.

Handles to Anonymous Functions

Function handles also serve as the means of invoking anonymous functions. An anonymous function is a one-line expression-based MATLAB function that does not require an M-file.

For example, the statement

```
sqr = @(x) x.^2;
```

creates an anonymous function that computes the square of its input argument `x`. The @ operator makes `sqr` a function handle, giving you a means of calling the function:

```
sqr(20)
ans =
   400
```

See “Anonymous Functions” on page 4-3 for more information on anonymous functions.

Function Handle Arrays

To create an array of function handles, you must use a cell array:

```
trigFun = {@sin, @cos, @tan};
```

For example, to plot the cosine of the range `-pi` to `pi` at 0.01 intervals, use

```
plot(trigFun{2}(-pi:0.01:pi))
```

Calling a Function Using Its Handle

To execute a function associated with a function handle, use the syntax shown here. In this example, `h` is a variable that has a function handle as its value:

```
h(arg1, arg2, ..., argn)
```

If the function being called takes no input arguments, then use empty parentheses after the handle name to call the function:

```
h()
```

If you use only the variable name, not followed by parentheses, MATLAB just identifies the name of the associated function and does not invoke the function:

```
h = @computer;    % Construct the handle h.

h
h =               % h alone just identifies the handle.
    @computer

h()               % h with parentheses invokes the function.
ans =
   PCWIN
```

Example of Passing a Function Handle

The following example creates a handle for a function supplied by MATLAB called `humps` and assigns it to the variable `h`. (The `humps` function returns a strong maxima near x = 0.3 and x = 0.9).

```
h = @humps;
```

After constructing the handle, you can pass it in the argument list of a call to some other function, as shown here. This example passes the function handle `h` that was just created as the first argument in a call to `fminbnd`. This function then minimizes over the interval [0.3, 1].

```
x = fminbnd(h, 0.3, 1)
x =
    0.6370
```

Using a function handle enables you to pass different functions for `fminbnd` to use in determining its final result.

Handling Values Returned From a Call

When you invoke a function using a function handle, you can capture any or all values returned from the call in the same way you would if you were calling the function directly. Just list the output variables to the left of the equals (=) sign. When assigning to multiple outputs, enclose the output variables within square brackets:

```
[out1 out2 ...] = h(arg1, arg2, arg3, ...)
```

The example below returns multiple values from a call to an anonymous function. Create anonymous function `f` that locates the nonzero elements of an array, and returns the row, column, and value of each element in variables `row`, `col`, and `val`:

```
f = @(X)find(X);
```

Call the function on matrix `m` using the function handle `f`. Because the function uses the MATLAB `find` function which returns up to three outputs, you can specify from 0 to 3 outputs in the call:

```
m = [3 2 0; -5 0 7; 0 0 1]
```

```
m =
     3     2     0
    -5     0     7
     0     0     1

[row col val] = f(m);

val
val =
     3
    -5
     2
     7
     1
```

Applications of Function Handles

The following sections discuss the advantages of using function handles:

- "Pass a Function to Another Function" on page 1-131
- "Capture Data Values For Later Use By a Function" on page 1-133
- "Call Functions Outside of Their Normal Scope" on page 1-135
- "Save the Handle in a MAT-File for Use in a Later MATLAB Session" on page 1-136

Pass a Function to Another Function

The ability to pass variables to a function enables you to run the function on different values. In the same way, you can pass function handles as input arguments to a function, thus allowing the called function to change the operations it runs on the input data.

Example 1 — Run quad on Varying Functions. Run the quadrature function on varying input functions:

```
a = 0;  b = 5;

quad(@log, a, b)
ans =
    3.0472
```

```
quad(@sin, a, b)
ans =
    0.7163

quad(@humps, a, b)
ans =
   12.3566
```

Example 2 — Run quad on Anonymous Functions. Run `quad` on a MATLAB built-in function or an anonymous function:

```
n = quad(@log, 0, 3);

n = quad(@(x)x.^2, 0, 3);
```

Change the parameters of the function you pass to `quad` with a simple modification of the anonymous function that is associated with the function handle input:

```
a = 3.7;
z = quad(@(x)x.^a, 0, 3);
```

Example 3 — Compare quad Results on Different Functions. Compare the integral of the cosine function over the interval `[a, b]`:

```
a = 0; b = 10;
int1 = quad(@cos,a,b)

int1 =
   -0.5440
```

with the integral over the same interval of the piecewise polynomial `pp` that approximates the cosine function by interpolating the computed values `x` and `y`:

```
x = a:b;
y = cos(x);
pp = spline(x,y);
int2 = quad(@(x)ppval(pp,x), a, b)
```

```
int2 =
   -0.5485
```

Capture Data Values For Later Use By a Function

You can do more with a function handle than just create an association to a certain function. By using anonymous functions, you can also capture certain variables and their values from the function workspace and store them in the handle. These data values are stored in the handle at the time of its construction, and are contained within the handle for as long as it exists. Whenever you then invoke the function by means of its handle, MATLAB supplies the function with all variable inputs specified in the argument list of the function call, and also any constant inputs that were stored in the function handle at the time of its construction.

Storing some or all input data in a function handle enables you to reliably use the same set of data with that function regardless of where or when you invoke the handle. You can also interrupt your use of a function and resume it with the same data at a later time simply by saving the function handle to a MAT-file.

Example 1 — Constructing a Function Handle to Hold Data. Compare the following two ways of implementing a simple plotting function called `draw_plot`. The first case creates the function as one that you would call by name and that accepts seven inputs specifying coordinate and property information:

```
function draw_plot(x, y, lnSpec, lnWidth, mkEdge, mkFace, mkSize)
plot(x, y, lnSpec, ...
        'LineWidth', lnWidth, ...
        'MarkerEdgeColor', mkEdge, ...
        'MarkerFaceColor', mkFace, ...
        'MarkerSize', mkSize)
```

The second case implements `draw_plot` as an anonymous function to be called by a function handle, `h`. The `draw_plot` function has only two inputs now; the remaining five are specified only on a call to the handle constructor function, `get_plot_handle`:

```
function h = get_plot_handle(lnSpec, lnWidth, mkEdge, ...
   mkFace, mkSize)
```

```
h = @draw_plot;
    function draw_plot(x, y)
        plot(x, y, lnSpec, ...
            'LineWidth', lnWidth, ...
            'MarkerEdgeColor', mkEdge, ...
            'MarkerFaceColor', mkFace, ...
            'MarkerSize', mkSize)
    end
end
```

Because these input values are required by the `draw_plot` function but are not made available in its argument list, MATLAB supplies them by storing them in the function handle for `draw_plot` at the time it is constructed. Construct the function handle `h`, also supplying the values to be stored in handle:

```
h = get_plot_handle('--rs', 2, 'k', 'g', 10);
```

Now call the function, specifying only the `x` and `y` inputs:

```
x = -pi:pi/10:pi;
y = tan(sin(x)) - sin(tan(x));
h(x, y)                               % Draw the plot
```

The later section on “Examining a Function Handle” on page 1-137 continues this example by showing how you can examine the contents of the function and workspace contents of this function handle.

Example 2 — Varying Data Values Stored in a Function Handle. Values stored within a handle to a nested function do not have to remain constant. The following function constructs and returns a function handle `h` to the anonymous function `addOne`. In addition to associating the handle with `addOne`, MATLAB also stores the initial value of `x` in the function handle:

```
function h = counter
x = 0;
h = @addOne;
   function y = addOne;
   x = x + 1;
   y = x;
   end
end
```

The addOne function that is associated with handle h increments variable x each time you call it. This modifies the value of the variable stored in the function handle:

```
h = counter;
h()
    1
h()
    2
```

Example 3 — You Cannot Vary Data in a Handle to an Anonymous Function. Unlike the example above, values stored within a handle to an anonymous function do remain constant. Construct a handle to an anonymous function that just returns the value of x, and initialize x to 300. The value of x within the function handle remains constant regardless of how you modify x external to the handle:

```
x = 300;
h = @()x;

x = 50;
h()
ans =
    300

clear x
h()
ans =
    300
```

Call Functions Outside of Their Normal Scope

By design, only functions within an M-file are permitted to access subfunctions defined within that file. However, if, in this same M-file, you were to construct a function handle for one of the internal subfunctions, and then pass that handle to a variable that exists outside of the M-file, access to that subfunction would be essentially unlimited. By capturing the access to the subfunction in a function handle, and then making that handle available to functions external to the M-file (or to the command line), the example extends

that scope. An example of this is shown in the preceding section, "Capture Data Values For Later Use By a Function" on page 1-133.

Private functions also have specific access rules that limit their availability with the MATLAB environment. But, as with subfunctions, MATLAB allows you to construct a handle for a private function. Therefore, you can call it by means of that handle from any location or even from the MATLAB command line, should it be necessary.

Save the Handle in a MAT-File for Use in a Later MATLAB Session

If you have one or more function handles that you would like to reuse in a later MATLAB session, you can store them in a MAT-file using the `save` function and then use `load` later on to restore them to your MATLAB workspace.

Saving and Loading Function Handles

You can save and load function handles in a MAT-file using the MATLAB `save` and `load` functions. If you load a function handle that you saved in an earlier MATLAB session, the following conditions could cause unexpected behavior:

- Any of the M-files that define the function have been moved, and thus no longer exist on the path stored in the handle.
- You load the function handle into an environment different from that in which it was saved. For example, the source for the function either doesn't exist or is located in a different directory than on the system on which the handle was saved.

In both of these cases, the function handle is now invalid because it is no longer associated with any existing function code. Although the handle is invalid, MATLAB still performs the load successfully and without displaying a warning. Attempting to invoke the handle, however, results in an error.

Invalid or Obsolete Function Handles

If you create a handle to a function that is not on the MATLAB path, or if you `load` a handle to a function that is no longer on the path, MATLAB catches the error only when the handle is invoked. You can assign an invalid handle

and use it in such operations as `func2str`. MATLAB catches and reports an error only when you attempt to use it in a runtime operation.

Advanced Operations on Function Handles

Advanced operations include:

- "Examining a Function Handle" on page 1-137
- "Converting to and from a String" on page 1-138
- "Comparing Function Handles" on page 1-139

Examining a Function Handle

Use the `functions` function to examine the contents of a function handle.

Caution MATLAB provides the `functions` function for querying and debugging purposes only. Because its behavior may change in subsequent releases, you should not rely upon it for programming purposes.

The following example is a continuation of the example shown in "Example 1 — Constructing a Function Handle to Hold Data" on page 1-133. The example constructs a function handle that contains both a function association, and data required by that function to execute. The function shown here constructs the function handle, `h`:

```
function h = get_plot_handle(lnSpec, lnWidth, mkEdge, ...
   mkFace, mkSize)
h = @draw_plot;
    function draw_plot(x, y)
        plot(x, y, lnSpec, ...
            'LineWidth', lnWidth, ...
            'MarkerEdgeColor', mkEdge, ...
            'MarkerFaceColor', mkFace, ...
            'MarkerSize', mkSize)
    end
end
```

Use `functions` to examine the contents of the returned handle:

```
f = functions(h)
f =
     function: 'get_plot_handle/draw_plot'
         type: 'nested'
         file: 'D:\matlab\work\get_plot_handle.m'
    workspace: {[1x1 struct]}
```

The call to `functions` returns a structure with four fields:

- `function` — Name of the function or subfunction to which the handle is associated. (Function names that follow a slash character (/) are implemented in the program code as subfunctions.)
- `type` — Type of function (e.g., simple, nested, anonymous)
- `file` — Filename and path to the M-file. (For built-in functions, this is the string `'MATLAB built-in function'`)
- `workspace` — Variables in the function workspace at the time the handle was constructed, along with their values

Examine the workspace variables that you saved in the function handle:

```
f.workspace{:}
ans =
            h: @get_plot_handle/draw_plot
       lnSpec: '--rs'
      lnWidth: 2
     mrkrEdge: 'k'
     mrkrFace: 'g'
     mrkrSize: 10
```

Converting to and from a String

Two functions, `str2func` and `func2str` enable you to convert between a string containing a function name and a function handle that is associated with that function name.

Converting a String to a Function Handle. Another means of creating a function handle is to convert a string that holds a function name to a handle for the named function. You can do this using the `str2func` function:

```
handle = str2func('functionname');
```

The example below takes the name of a function as the first argument. It compares part of the name to see if this is a polynomial function, converts the function string to a function handle if it is not, and then calls the function by means of its handle:

```
function run_function(funcname, arg1, arg2)
if strncmp(funcname, 'poly', 4)
    disp 'You cannot run polynomial functions on this data.'
    return
else
   h = str2func(funcname);
   h(arg1, arg2);
end
```

Note Nested functions are not accessible to `str2func`. To construct a function handle for a nested function, you must use the function handle constructor, `@`.

Converting a Function Handle to a String. You can also convert a function handle back into a string using the `func2str` function:

```
functionname = func2str(handle);
```

This example converts the function handle `h` to a string containing the function name, and then uses the function name in a message displayed to the user:

```
function call_h(h, arg1, arg2)
sprintf('Calling function %s ...\n', func2str(h))
h(arg1, arg2)
```

Comparing Function Handles

This section describes how MATLAB determines whether or not separate function handles are equal to each other:

- "Comparing Handles Constructed from a Named Function" on page 1-140

- "Comparing Handles to Anonymous Functions" on page 1-140
- "Comparing Handles to Nested Functions" on page 1-141
- "Comparing Handles Saved to a MAT-File" on page 1-142

Comparing Handles Constructed from a Named Function. MATLAB considers function handles that you construct from the same named function (e.g., `handle = @sin`) to be equal. The `isequal` function returns a value of `true` when comparing these types of handles:

```
func1 = @sin;
func2 = @sin;
isequal(func1, func2)
 ans =
     1
```

If you save these handles to a MAT-file, and then load them back into the workspace later on, they are still equal.

Comparing Handles to Anonymous Functions. Unlike handles to named functions, any two function handles that represent the same anonymous function (i.e., handles to anonymous functions that contain the same text) are not equal. This is because MATLAB cannot guarantee that the frozen values of non-argument variables (such as `A`, below) are the same.

```
A = 5;
h1 = @(x)A * x.^2;
h2 = @(x)A * x.^2;

isequal(h1, h2)
ans =
     0
```

Note In general, MATLAB may underestimate the equality of function handles. That is, a test for equality may return `false` even when the functions happen to behave the same. But in cases where MATLAB does indicate equality, the functions are guaranteed to behave in an identical manner.

If you make a copy of an anonymous function handle, the copy and the original are equal:

```
h1 = @(x)A * x.^2;   h2 = h1;
isequal(h1, h2)
ans =
     1
```

Comparing Handles to Nested Functions. MATLAB considers function handles to the same nested function to be equal only if your code constructs these handles on the same call to the function containing the nested functions. Given this function that constructs two handles to the same nested function:

```
function [h1, h2] = test_eq(a, b, c)
h1 = @findZ;
h2 = @findZ;

   function z = findZ
   z = a.^3 + b.^2 + c';
   end
end
```

function handles constructed from the same nested function and on the same call to the parent function are considered equal:

```
[h1 h2] = test_eq(4, 19, -7);

isequal(h1, h2),
ans =
     1
```

while those constructed from different calls are not considered equal:

```
[q1 q2] = test_eq(3, -1, 2);

isequal(h1, q1)
ans =
     0
```

Comparing Handles Saved to a MAT-File. If you save equivalent anonymous or nested function handles to separate MAT-files, and then load them back into the MATLAB workspace, they are no longer equal. This is because saving the function handle loses track of the original circumstances under which the function handle was created. Reloading it results in a function handle that compares as being unequal to the original function handle.

Create two equivalent anonymous function handles:

```
h1 = @(x) sin(x);
h2 = h1;

isequal(h1, h2)
ans =
     1
```

Save each to a different MAT-file:

```
save fname1 h1;
save fname2 h2;
```

Clear the MATLAB workspace, and then load the function handles back into the workspace:

```
clear all
load fname1
load fname2
```

The function handles are no longer considered equal:

```
isequal(h1, h2)
ans =
     0
```

Note, however, that equal anonymous and nested function handles that you save to the same MAT-file are equal when loaded back into MATLAB.

Functions That Operate on Function Handles

MATLAB provides the following functions for working with function handles. See the reference pages for these functions for more information.

Function	Description
`functions`	Return information describing a function handle.
`func2str`	Construct a function name string from a function handle.
`str2func`	Construct a function handle from a function name string.
`save`	Save a function handle from the current workspace to a MAT-file.
`load`	Load a function handle from a MAT-file into the current workspace.
`isa`	Determine if a variable contains a function handle.
`isequal`	Determine if two function handles are handles to the same function.

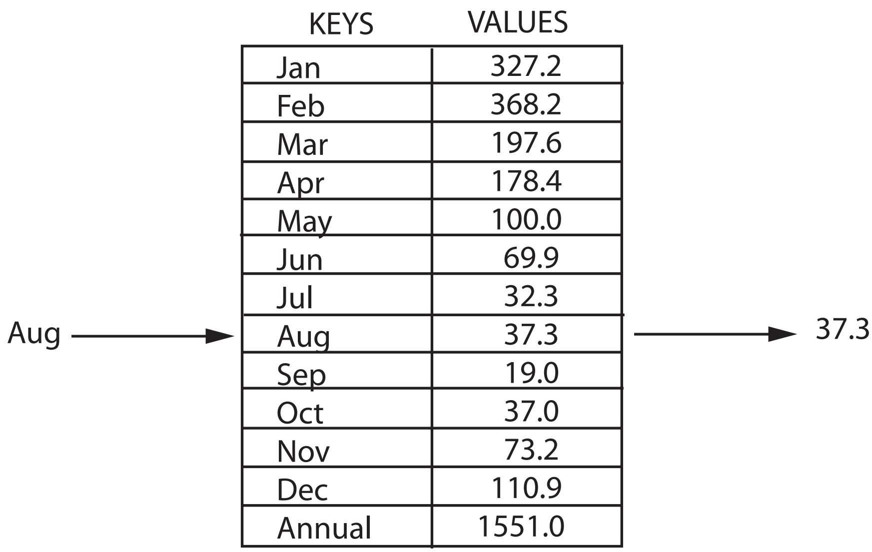
Map Containers

In this section...

Overview of the Map Data Structure

A *Map* is a type of fast key lookup data structure that offers a flexible means of indexing into its individual elements. Unlike most array data structures in the MATLAB software that only allow access to the elements by means of integer indices, indices for a Map can be nearly any scalar numeric value or a character string.

Indices into the elements of a Map are called *keys*. These keys, along with the data *values* associated with them, are stored within the Map. Each entry of a Map contains exactly one unique key and its corresponding value. Indexing into the Map of rainfall statistics shown below with a string representing the month of August yields the value internally associated with that month, 37.3.



Mean monthly rainfall statistics (mm)

Keys are not restricted to integers as they are with other arrays. Specifically, a key may be any of the following types:

- 1-by-N character array
- Scalar real `double` or `single`
- Signed or unsigned scalar integer

The values stored in a Map can be of any type. This includes arrays of numeric values, structures, cells, strings, objects, or other Maps.

Note A Map is most memory efficient when the data stored in it is a scalar number or a character array.

Description of the Map Class

A Map is actually an object, or instance, of a MATLAB class called `Map`. It is also a handle object and, as such, it behaves like any other MATLAB handle object. This section gives a brief overview of the `Map` class. For more details,

see the function reference pages for the Map constructor or for any method of the class.

Properties of the Map Class

All objects of the Map class have three properties. You cannot write directly to any of these properties; you can change them only by means of the methods of the Map class.

Property	Description	Default
Count	Unsigned 64-bit integer that represents the total number of key/value pairs contained in the Map object.	0
KeyType	String that indicates the type of all keys contained in the Map object. KeyType can be any of the following: double, single, char, and signed or unsigned 32-bit or 64-bit integer. If you attempt to add keys of an unsupported type, int8 for example, MATLAB makes them double.	char
ValueType	String that indicates the type of values contained in the Map object. If the values in a Map are all scalar numbers of the same type, ValueType is set to that type. If the values are all character arrays, ValueType is 'char'. Otherwise, ValueType is 'any'.	any

To examine one of these properties, follow the name of the Map object with a dot and then the property name. For example, to see what type of keys are used in Map mapObj, use

```
mapObj.KeyType
```

A Map is a handle object. As such, if you make a copy of the object, MATLAB does not create a new Map; it creates a new handle for the existing Map that you specify. If you alter the Map's contents in reference to this new handle, MATLAB applies the changes you make to the original Map as well. You can, however, delete the new handle without affecting the original Map.

Methods of the Map Class

The `Map` class implements the following methods. Their use is explained in the later sections of this documentation and also in the function reference pages.

Method	Description
`isKey`	Check if Map contains specified key
`keys`	Names of all keys in Map
`length`	Length of Map
`remove`	Remove key and its value from Map
`size`	Dimensions of Map
`values`	Values contained in Map

Creating a Map Object

A Map is an object of the `Map` class. It is defined within a MATLAB package called `containers`. As with any class, you use its constructor function to create any new instances of it. You must include the package name when calling the constructor:

```
newMap = containers.Map(optional_keys_and_values)
```

Constructing an Empty Map Object

When you call the Map constructor with no input arguments, MATLAB constructs an empty `Map` object. When you do not end the command with a semicolon, MATLAB displays the following information about the object you have constructed:

```
  newMap = containers.Map()
newMap =
   containers.Map handle
   package: containers
   properties:
           Count: 0
         KeyType: 'char'
       ValueType: 'any'
   lists of Methods, Events, Superclasses
```

The properties of an empty `Map` object are set to their default values:

- `Count = 0`
- `KeyType = 'char'`
- `ValueType = 'any'`

Once you construct the empty Map object, you can use the `keys` and `values` methods to populate it. For a summary of MATLAB functions you can use with a Map object, see "Methods of the Map Class" on page 1-147

Constructing An Initialized Map Object

Most of the time, you will want to initialize the Map with at least some keys and values at the time you construct it. You can enter one or more sets of keys and values using the syntax shown here. The brace operators ({}) are not required if you enter only one key/value pair:

```
mapObj = containers.Map({key1, key2, ...}, {val1, val2, ...});
```

For those keys and values that are character strings, be sure that you specify them enclosed within single quotation marks. For example, when constructing a Map that has character string keys, use

```
mapObj = containers.Map(...
   {'keystr1', 'keystr2', ...}, {val1, val2, ...});
```

As an example of constructing an initialized `Map` object, create a new Map for the following key/value pairs taken from the monthly rainfall map shown earlier in this section.

KEYS	VALUES
Jan	327.2
Feb	368.2
Mar	197.6
Apr	178.4
May	100.0
Jun	69.9
Jul	32.3
Aug	37.3
Sep	19.0
Oct	37.0
Nov	73.2
Dec	110.9
Annual	1551.0

```
k = {'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', ...
  'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec', 'Annual'};

v = {327.2, 368.2, 197.6, 178.4, 100.0,  69.9, ...
  32.3,  37.3,  19.0,  37.0,  73.2, 110.9, 1551.0};

rainfallMap = containers.Map(k, v)
rainfallMap =
  containers.Map handle
  Package: containers

  Properties:
        Count: 13
      KeyType: 'char'
    ValueType: 'double'
  Methods, Events, Superclasses
```

The `Count` property is now set to the number of key/value pairs in the Map, `13`, the `KeyType` is `char`, and the `ValueType` is `double`.

Combining Map Objects

You can combine `Map` objects vertically using concatenation. However, the result is not a vector of Maps, but rather a single `Map` object containing all key/value pairs of the contributing Maps. Horizontal vectors of Maps are not allowed. See "Building a Map with Concatenation" on page 1-153, below.

Examining the Contents of the Map

Each entry in a Map consists of two parts: a unique key and its corresponding value. To find all the keys in a Map, use the `keys` method. To find all of the values, use the `values` method.

Create a new Map called `tickets` that maps airline ticket numbers to the holders of those tickets. Construct the Map with four key/value pairs:

```
ticketMap = containers.Map(...
    {'2R175', 'B7398', 'A479GY', 'NZ1452'}, ...
    {'James Enright', 'Carl Haynes', 'Sarah Latham', ...
     'Bradley Reid'});
```

Use the `keys` method to display all keys in the Map. MATLAB lists keys of type `char` in alphabetical order, and keys of any numeric type in numerical order:

```
keys(ticketMap)
ans =
   '2R175'   'A479GY'   'B7398'   'NZ1452'
```

Next, display the values that are associated with those keys in the Map. The order of the values is determined by the order of the keys associated with them.

This table shows the keys listed in alphabetical order:

keys	values
2R175	James Enright
A479GY	Sarah Latham

keys	values
B7398	Carl Haynes
NZ1452	Bradley Reid

The `values` method uses the same ordering of values:

```
values(ticketMap)
ans =
  'James Enright'  'Sarah Latham'  'Carl Haynes'  'Bradley Reid'
```

Reading and Writing Using a Key Index

When reading from the Map, use the same keys that you have defined and associated with particular values. Writing new entries to the Map requires that you supply the values to store with a key for each one .

Note For a large Map, the keys and value methods use a lot of memory as their outputs are cell arrays.

Reading From the Map

After you have constructed and populated your Map, you can begin to use it to store and retrieve data. You use a Map in the same manner that you would an array, except that you are not restricted to using integer indices. The general syntax for looking up a value (`valueN`) for a given key (`keyN`) is shown here. If the key is a character string, enclose it in single quotation marks:

```
valueN = mapObj(keyN);
```

You can find any single value by indexing into the map with the appropriate key:

```
passenger = ticketMap('2R175')
passenger =
   James Enright
```

Find the person who holds ticket `A479GY`:

```
sprintf('   Would passenger %s please come to the desk?\n', ...
    ticketMap('A479GY'))
ans =
    Would passenger Sarah Latham please come to the desk?
```

To access the values of multiple keys, use the `values` method, specifying the keys in a cell array:

```
values(ticketMap, {'2R175', 'B7398'})
ans =
    'James Enright'    'Carl Haynes'
```

You cannot use the colon operator to access a range of keys as you can with other MATLAB classes. For example, the following statement throws an error:

```
ticketMap('2R175':'B7398')
```

Adding Key/Value Pairs

Unlike other array types, each entry in a Map consists of two items: the value and its key. When you write a new value to a Map, you must supply its key as well. This key must be consistent in type with any other keys in the Map.

Use the following syntax to insert additional elements into a Map:

```
existingMapObj(newKeyName) = newValue;
```

Add two more entries to the `ticketMap` used in the above examples, Verify that the Map now has five key/value pairs:

```
ticketMap('947F4') = 'Susan Spera';
ticketMap('417R93') = 'Patricia Hughes';

ticketMap.Count
ans =
     6
```

List all of the keys and values in Map `ticketMap`:

```
keys(ticketMap),  values(ticketMap)
ans =
```

```
  '2R175'   '417R93'   '947F4'   'A479GY'   'B7398'   'NZ1452'
ans =
  Columns 1 through 3
    'James Enright'   'Patricia Hughes'   'Susan Spera'
  Columns 4 through 6
    'Sarah Latham'   'Carl Haynes'   'Bradley Reid'
```

Building a Map with Concatenation

You can add key/value pairs to a Map in groups using concatenation. The concatenation of Map objects is different from other classes. Instead of building a vector of s, MATLAB returns a single Map containing the key/value pairs from each of the contributing Map objects.

Rules for the concatenation of Map objects are:

- Only vertical vectors of Map objects are allowed. You cannot create an m-by-n array or a horizontal vector of s. For this reason, `vertcat` is supported for Map objects, but not `horzcat`.
- All keys in each map being concatenated must be of the same class.
- You can combine Maps with different numbers of key/value pairs. The result is a single Map object containing key/value pairs from each of the contributing maps:

  ```
  tMap1 = containers.Map({'2R175', 'B7398', 'A479GY'}, ...
      {'James Enright', 'Carl Haynes', 'Sarah Latham'});

  tMap2 = containers.Map({'417R93', 'NZ1452', '947F4'}, ...
      {'Patricia Hughes', 'Bradley Reid', 'Susan Spera'});

  % Concatenate the two maps:
  ticketMap = [tMap1; tMap2];
  ```

  The result of this concatenation is the same 6-element map that was constructed in the previous section:

  ```
  ticketMap.Count
  ans =
      6
  ```

```
keys(ticketMap),  values(ticketMap)
ans =
   '2R175'  '417R93'  '947F4'  'A479GY'  'B7398'  'NZ1452'
ans =
  Columns 1 through 3
    'James Enright'    'Patricia Hughes'    'Susan Spera'
  Columns 4 through 6
    'Sarah Latham'    'Carl Haynes'    'Bradley Reid'
```

- Concatenation does not include duplicate keys or their values in the resulting Map object.

  In the following example, both objects `m1` and `m2` use a key of `8`. In Map `m1`, `8` is a key to value `C`; in `m2`, it is a key to value `X`:

  ```
  m1 = containers.Map({1, 5, 8}, {'A', 'B', 'C'});
  m2 = containers.Map({8, 9, 6}, {'X', 'Y', 'Z'});
  ```

  Combine `m1` and `m2` to form a new Map object, `m`:

  ```
  m = [m1; m2];
  ```

  The resulting Map object `m` has only five key/value pairs. The value C was dropped from the concatenation because its key was not unique:

  ```
  keys(m), values(m)
  ans =
      [1]    [5]    [6]    [8]    [9]
  ans =
      'A'    'B'    'Z'    'X'    'Y'
  ```

Modifying Keys and Values in the Map

In addition to reading and writing the contents of a Map, you can also delete key/value pairs and modify any of its values or keys.

Note Keep in mind that if you have more than one handle to a Map, modifying the handle also makes changes to the original Map. See "Modifying a Copy of the Map" on page 1-156, below.

Removing Keys and Values from the Map

Use the `remove` method to delete any entries from a Map. When calling this method, specify the `Map` object name and the key name to remove. MATLAB deletes the key and its associated value from the Map.

The syntax for the remove method is

```
remove('mapName', 'keyname');
```

Remove one entry (the specified key and its value) from the `Map` object:

```
remove(ticketMap, 'NZ1452');
values(ticketMap)
ans =
  Columns 1 through 3
    'James Enright'    'Patricia Hughes'    'Susan Spera'
  Columns 4 through 5
    'Sarah Latham'    'Carl Haynes'
```

Modifying Values

You can modify any value in a Map simply by overwriting the current value. The passenger holding ticket `A479GY` is identified as `Sarah Latham`:

```
ticketMap('A479GY')
ans =
    Sarah Latham
```

Change the passenger's first name to `Anna Latham` by overwriting the original value for the `A479GY` key:

```
ticketMap('A479GY') = 'Anna Latham';
```

Verify the change:

```
ticketMap('A479GY')
ans =
    'Anna Latham';
```

Modifying Keys

To modify an existing key while keeping the value the same, first remove both the key and its value from the Map. Then create a new entry, this time with the corrected key name.

Modify the ticket number belonging to passenger James Enright:

```
remove(ticketMap, '2R175');
ticketMap('2S185') = 'James Enright';

k = keys(ticketMap);  v = values(ticketMap);
str1 = '   ''%s'' has been assigned a new\n';
str2 = '    ticket number: %s.\n';

fprintf(str1, v{1})
fprintf(str2, k{1})

 'James Enright' has been assigned a new
    ticket number: 2S185.
```

Modifying a Copy of the Map

Because `ticketMap` is a handle object, you need to be careful when making copies of the Map. Keep in mind that by copying a `Map` object, you are really just creating another handle to the same object. Any changes you make to this handle are also applied to the original Map.

Make a copy of Map `ticketMap`. Write to this copy, and notice that the change is applied to the original Map object itself:

```
copiedMap = ticketMap;

copiedMap('AZ12345') = 'unidentified person';
ticketMap('AZ12345')
ans =
    unidentified person
```

Clean up:

```
remove(ticketMap, 'AZ12345');
```

```
clear copiedMap;
```

Mapping to Different Value Types

It is fairly common to store other classes, such as structures or cell arrays, in a Map structure. However, Maps are most memory efficient when the data stored in them belongs to one of the basic MATLAB types such as double, char, integers, and logicals.

Mapping to a Structure Array

The following example maps airline seat numbers to structures that contain information on who occupies the seat. To start out, create the following structure array:

```
s1.ticketNum = '2S185'; s1.destination = 'Barbados';
s1.reserved = '06-May-2008'; s1.origin = 'La Guardia';
s2.ticketNum = '947F4'; s2.destination = 'St. John';
s2.reserved = '14-Apr-2008'; s2.origin = 'Oakland';
s3.ticketNum = 'A479GY'; s3.destination = 'St. Lucia';
s3.reserved = '28-Mar-2008'; s3.origin = 'JFK';
s4.ticketNum = 'B7398'; s4.destination = 'Granada';
s4.reserved = '30-Apr-2008'; s4.origin = 'JFK';
s5.ticketNum = 'NZ1452'; s5.destination = 'Aruba';
s5.reserved = '01-May-2008'; s5.origin = 'Denver';
```

Map five of the seats to one of these structures:

```
seatingMap = containers.Map( ...
    {'23F', '15C', '15B', '09C', '12D'}, ...
    {s5, s1, s3, s4, s2});
```

Using this Map object, find information about the passenger, who has reserved seat 09C:

```
seatingMap('09C')
ans =
      ticketNum: 'B7398'
    destination: 'Granada'
       reserved: '30-Apr-2008'
         origin: 'JFK'
```

```
seatingMap('15B').ticketNum
ans =
    A479GY
```

Using two Maps together, you can find out the name of the person who has reserved the seat:

```
passenger = ticketMap(seatingMap('15B').ticketNum)
passenger =
   Anna Latham
```

Mapping to a Cell Array

As with structures, you can also map to a cell array in a Map object. Continuing with the airline example of the previous sections, some of the passengers on the flight have "frequent flyer" accounts with the airline. Map the names of these passengers to records of the number of miles they have used and the number of miles they still have available:

```
accountMap = containers.Map( ...
    {'Susan Spera','Carl Haynes','Anna Latham'}, ...
    {{247.5, 56.1}, {0, 1342.9}, {24.6, 314.7}});
```

Use the Map to retrieve account information on the passengers:

```
name = 'Carl Haynes';
acct = accountMap(name);

fprintf('%s has used %.1f miles on his/her account,\n', ...
    name, acct{1})
fprintf('  and has %.1f miles remaining.\n', acct{2})

Carl Haynes has used 0.0 miles on his/her account,
  and has 1342.9 miles remaining.
```

Combining Unlike Classes

In this section...

Matrices and arrays can be composed of elements of most any MATLAB data type as long as all elements in the matrix are of the same type. If you do include elements of unlike classes when constructing a matrix, MATLAB converts some elements so that all elements of the resulting matrix are of the same type. (See Chapter 1, "Classes (Data Types)" for information on any of the MATLAB classes discussed here.)

Data type conversion is done with respect to a preset precedence of classes. The following table shows the five classes you can concatenate with an unlike type without generating an error (that is, with the exception of character and logical).

TYPE	character	integer	single	double	logical
character	character	character	character	character	invalid
integer	character	integer	integer	integer	integer
single	character	integer	single	single	single
double	character	integer	single	double	double
logical	invalid	integer	single	double	logical

For example, concatenating a `double` and `single` matrix always yields a matrix of type `single`. MATLAB converts the `double` element to `single` to accomplish this.

Combining Unlike Integer Types

If you combine different integer types in a matrix (e.g., signed with unsigned, or 8-bit integers with 16-bit integers), MATLAB returns a matrix in which all elements are of one common type. MATLAB sets all elements of the resulting matrix to the data type of the left-most element in the input matrix. For example, the result of the following concatenation is a vector of three 16-bit signed integers:

```
A = [int16(450) uint8(250) int32(1000000)]
```

MATLAB also displays a warning to inform you that the result may not be what you had expected:

```
A = [int16(450) uint8(250) int32(1000000)];
Warning: Concatenation with dominant (left-most) integer class
may overflow other operands on conversion to return class.
```

You can disable this warning by entering the following two commands directly after the operation that caused the warning. The first command retrieves the message identifier associated with the most recent warning issued by MATLAB. The second command uses this identifier to disable any further warnings of that type from being issued:

```
[msg, intcat_msgid] = lastwarn;
warning('off', intcat_msgid);
```

To reenable the warning so that it will now be displayed, use

```
warning('on', intcat_msgid);
```

You can use these commands to disable or enable the display of any MATLAB warning.

Example of Combining Unlike Integer Sizes

After disabling the integer concatenation warnings as shown above, concatenate the following two numbers once, and then switch their order. The return value depends on the order in which the integers are concatenated. The left-most type determines the data type for all elements in the vector:

```
A = [int16(5000) int8(50)]
```

```
A =
   5000   50

B = [int8(50) int16(5000)]
B =
   50   127
```

The first operation returns a vector of 16-bit integers. The second returns a vector of 8-bit integers. The element `int16(5000)` is set to `127`, the maximum value for an 8-bit signed integer.

The same rules apply to vertical concatenation:

```
C = [int8(50); int16(5000)]
C =
    50
   127
```

Note You can find the maximum or minimum values for any MATLAB integer type using the `intmax` and `intmin` functions. For floating-point types, use `realmax` and `realmin`.

Example of Combining Signed with Unsigned

Now do the same exercise with signed and unsigned integers. Again, the left-most element determines the data type for all elements in the resulting matrix:

```
A = [int8(-100) uint8(100)]
A =
   -100   100

B = [uint8(100) int8(-100)]
B =
   100   0
```

The element `int8(-100)` is set to zero because it is no longer signed.

MATLAB evaluates each element *prior to* concatenating them into a combined array. In other words, the following statement evaluates to an 8-bit signed integer (equal to 50) and an 8-bit unsigned integer (unsigned -50 is set to zero) before the two elements are combined. Following the concatenation, the second element retains its zero value but takes on the unsigned `int8` type:

```
A = [int8(50), uint8(-50)]
A =
  50    0
```

Combining Integer and Noninteger Data

If you combine integers with `double`, `single`, or `logical` classes, all elements of the resulting matrix are given the data type of the left-most integer. For example, all elements of the following vector are set to `int32`:

```
A = [true pi int32(1000000) single(17.32) uint8(250)]
```

Empty Matrices

If you construct a matrix using empty matrix elements, the empty matrices are ignored in the resulting matrix:

```
A = [5.36; 7.01; []; 9.44]
A =
    5.3600
    7.0100
    9.4400
```

Concatenation Examples

Here are some examples of data type conversion during matrix construction.

Combining Single and Double Types

Combining `single` values with `double` values yields a `single` matrix. Note that `5.73*10^300` is too big to be stored as a `single`, thus the conversion from `double` to `single` sets it to infinity. (The `class` function used in this example returns the data type for the input value).

```
x = [single(4.5) single(-2.8) pi 5.73*10^300]
x =
    4.5000   -2.8000    3.1416       Inf
```

```
class(x)              % Display the data type of x
ans =
   single
```

Combining Integer and Double Types

Combining integer values with `double` values yields an integer matrix. Note that the fractional part of `pi` is rounded to the nearest integer. (The `int8` function used in this example converts its numeric argument to an 8-bit integer).

```
x = [int8(21) int8(-22) int8(23) pi 45/6]
x =
    21   -22    23     3     7

class(x)
ans =
   int8
```

Combining Character and Double Types

Combining `character` values with `double` values yields a `character` matrix. MATLAB converts the `double` elements in this example to their `character` equivalents:

```
x = ['A' 'B' 'C' 68 69 70]
x =
   ABCDEF

class(x)
ans =
   char
```

Combining Logical and Double Types

Combining `logical` values with `double` values yields a `double` matrix. MATLAB converts the `logical` `true` and `false` elements in this example to `double`:

```
x = [true false false pi sqrt(7)]
x =
    1.0000         0         0    3.1416    2.6458

class(x)
ans =
   double
```

Defining Your Own Classes

All MATLAB data types are implemented as object-oriented classes. You can add data types of your own to your MATLAB environment by creating additional classes. These user-defined classes define the structure of your new data type, and the M-file functions, or *methods*, that you write for each class define the behavior for that data type.

These methods can also define the way various MATLAB operators, including arithmetic operations, subscript referencing, and concatenation, apply to the new data types. For example, a class called `polynomial` might redefine the addition operator (+) so that it correctly performs the operation of addition on polynomials.

With MATLAB classes you can

- Create methods that override existing MATLAB functionality
- Restrict the operations that are allowed on an object of a class
- Enforce common behavior among related classes by inheriting from the same parent class
- Significantly increase the reuse of your code

Read more about MATLAB classes in the MATLAB Classes and Object-Oriented Programming documentation.

2

Basic Program Components

MATLAB Commands

In this section...

Basic Command Syntax

A simple MATLAB command computes the result of the expression to the right of the equals sign and assigns the value of the result to the output variable at the left. Examples of simple MATLAB commands are

```
x = 5.71;
A = [1 2 3; 4 5 6; 7 8 9];
I = besseli(nu, Z);
```

Commands that do not terminate with a semicolon display the result at your terminal as well as assigning it to the output variable:

```
A = [1 2 3; 4 5 6; 7 8 9]

A =
     1     2     3
     4     5     6
     7     8     9
```

If you do not explicitly assign the output of a command to a variable, MATLAB assigns the result to the reserved word `ans`:

```
[1 2 3; 4 5 6; 7 8 9]

ans =
     1     2     3
     4     5     6
     7     8     9
```

The value of `ans` changes with every command that returns an output value that is not assigned to a variable. We recommend that you do not use `ans` in place of variables as this practice tends to lead to code that is difficult to maintain and also to programming errors. The following use of `ans`, for example, is not recommended:

```
rand(3,5);
A = ans > 0.5;
```

Entering More Than One Command on a Line

You can enter more than one command on the same line, provided that you terminate each command with a comma or semicolon. Commands terminated with a comma display their results when they are executed; commands terminated with a semicolon do not:

```
rand('state', 0);
A = rand(3,5),   B = ones(3,5) * 4.7;   C = A./B

A =
    0.9501    0.4860    0.4565    0.4447    0.9218
    0.2311    0.8913    0.0185    0.6154    0.7382
    0.6068    0.7621    0.8214    0.7919    0.1763
C =
    0.2022    0.1034    0.0971    0.0946    0.1961
    0.0492    0.1896    0.0039    0.1309    0.1571
    0.1291    0.1621    0.1748    0.1685    0.0375
```

Assigning to Multiple Outputs

When a command generates more than one output, specify each output in square brackets to the left of the equals (=) sign. For example, the `deal` function distributes the values of each of its inputs to separate output variables:

```
[A B C] = deal([-12.3 4.89 -3.01], pi*1.46, diag(12:4:24))
A =
   -12.3000    4.8900   -3.0100
B =
    4.5867
C =
    12      0      0      0
```

```
     0    16     0     0
     0     0    20     0
     0     0     0    24
```

Other types of commands that can yield multiple outputs are assignments of structure and cell arrays, and calls to multiple-output functions. This example generates four outputs and assigns them to separate variables:

```
A(1).sym='H';  A(2).sym='He';  A(3).sym='Li';  A(4).sym='Be';
[hydrogen helium lithium beryllium] = A.sym

hydrogen =
   H
helium =
   He
lithium =
   Li
beryllium =
   Be
```

This example calls the `fileparts` function that returns up to four outputs, and assigns each output to a variable:

```
[path file exten vers] = ...
    fileparts('C:\matlab\work\strarray.mat')

path =
   C:\matlab\work
file =
   strarray
exten =
   .mat
vers =
     ''
```

For more information on assigning structure and cell arrays to multiple outputs, see "Assigning Struct Values to Separate Variables" on page 1-86 and "Assigning Cell Values to Separate Variables" on page 1-116. For information on assigning functions to multiple outputs, see "Assigning Output Arguments" on page 3-43.

Assigning Fewer Than the Full Number of Outputs

When assigning structure and cell arrays or when calling multiple-output functions, if you specify fewer output variables than there are return values, MATLAB assigns one return value to each output variable specified and discards the rest. Repeat the last two examples shown above, but specify fewer than the full number of outputs that are available:

```
[Symbol_1 Symbol_2] = A.symb
Symbol_1 =
   H
Symbol_2 =
   He


[path file] = fileparts('..\work\strarray.mat')
path =
   C:\matlab\work
file =
   strarray
```

The `deal` function , however, does require that you specify the full number of output variables.

Commands that Call MATLAB Functions

When entering commands that call functions in MATLAB, you can use either of two syntaxes: *command* or *function* syntax. This is explained in the section "Command vs. Function Syntax" on page 3-25 in the MATLAB Programming Fundamentals documentation.

Expressions

In this section...

String Evaluation

String evaluation adds power and flexibility to the MATLAB language, letting you perform operations like executing user-supplied strings and constructing executable strings through concatenation of strings stored in variables.

eval

The `eval` function evaluates a string that contains a MATLAB expression, statement, or function call. In its simplest form, the `eval` syntax is

```
eval('string')
```

For example, this code uses `eval` on an expression to generate a Hilbert matrix of order `n`.

```
t = '1/(m + n - 1)';
for m = 1:k
    for n = 1:k
        a(m,n) = eval(t);
    end
end
```

Here is an example that uses `eval` on a statement.

```
eval('t = clock');
```

Constructing Strings for Evaluation. You can concatenate strings to create a complete expression for input to `eval`. This code shows how `eval` can create 10 variables named `P1`, `P2`, `...`, `P10`, and set each of them to a different value.

```
for n = 1:10
    eval(['P', int2str(n), '= n .^ 2'])
```

```
end
```

feval

The `feval` function differs from `eval` in that it executes a function rather than a MATLAB expression. The function to be executed is specified in the first argument by either a function handle or a string containing the function name.

You can use `feval` and the `input` function to choose one of several tasks defined by M-files. This example uses function handles for the `sin`, `cos`, and `log` functions.

```
fun = {@sin; @cos; @log};
k = input('Choose function number: ');
x = input('Enter value: ');
feval(fun{k}, x)
```

Shell Escape Functions

It is sometimes useful to access your own C or Fortran programs using *shell escape functions*. Shell escape functions use the shell escape command `!` to make external stand-alone programs act like new MATLAB functions. A shell escape M-function is an M-file that

1. Saves the appropriate variables on disk.

2. Runs an external program (which reads the data file, processes the data, and writes the results back out to disk).

3. Loads the processed file back into the workspace.

For example, look at the code for `garfield.m`, below. This function uses an external function, `gareqn`, to find the solution to Garfield's equation.

```
function y = garfield(a,b,q,r)
save gardata a b q r
!gareqn
load gardata
```

This M-file

1. Saves the input arguments `a`, `b`, `q`, and `r` to a MAT-file in the workspace using the `save` command.

2. Uses the shell escape operator to access a C or Fortran program called `gareqn` that uses the workspace variables to perform its computation. `gareqn` writes its results to the `gardata` MAT-file.

3. Loads the `gardata` MAT-file described in “Using MAT-Files” to obtain the results.

Variables

In this section...

Types of Variables

A MATLAB variable is essentially a tag that you assign to a value while that value remains in memory. The tag gives you a way to reference the value in memory so that your programs can read it, operate on it with other data, and save it back to memory.

MATLAB provides three basic types of variables:

- "Local Variables" on page 2-9
- "Global Variables" on page 2-10
- "Persistent Variables" on page 2-12

Local Variables

Each MATLAB function has its own local variables. These are separate from those of other functions (except for nested functions), and from those of the base workspace. Variables defined in a function do not remain in memory from one function call to the next, unless they are defined as `global` or `persistent`.

Scripts, on the other hand, do not have a separate workspace. They store their variables in a workspace that is shared with the caller of the script. When called from the command line, they share the base workspace. When called from a function, they share that function's workspace.

Note If you run a script that alters a variable that already exists in the caller's workspace, that variable is overwritten by the script.

Global Variables

If several functions, and possibly the base workspace, all declare a particular name as `global`, then they all share a single copy of that variable. Any assignment to that variable, in any function, is available to all the other functions declaring it global.

Suppose, for example, you want to study the effect of the interaction coefficients, α and β, in the Lotka-Volterra predator-prey model.

$$\dot{y}_1 = y_1 - \alpha y_1 y_2$$

$$\dot{y}_2 = -y_2 + \beta y_1 y_2$$

Create an M-file, `lotka.m`.

```
function yp = lotka(t,y)
%LOTKA   Lotka-Volterra predator-prey model.
global ALPHA BETA
yp = [y(1) - ALPHA*y(1)*y(2); -y(2) + BETA*y(1)*y(2)];
```

Then interactively enter the statements

```
global ALPHA BETA
ALPHA = 0.01
BETA = 0.02
[t,y] = ode23(@lotka,[0,10],[1; 1]);
plot(t,y)
```

The two `global` statements make the values assigned to `ALPHA` and `BETA` at the command prompt available inside the function defined by `lotka.m`. They can be modified interactively and new solutions obtained without editing any files.

Creating Global Variables. Each function that uses a global variable must first declare the variable as `global`. It is usually best to put global declarations toward the beginning of the function. You would declare global variable `MAXLEN` as follows:

```
global MAXLEN
```

If the M-file contains subfunctions as well, then each subfunction requiring access to the global variable must declare it as `global`. To access the variable from the MATLAB command line, you must declare it as `global` at the command line.

MATLAB global variable names are typically longer and more descriptive than local variable names, and often consist of all uppercase characters. These are not requirements, but guidelines to increase the readability of MATLAB code, and to reduce the chance of accidentally redefining a global variable.

Displaying Global Variables. To see only those variables you have declared as global, use the `who` or `whos` functions with the literal, `global`.

```
global MAXLEN MAXWID
MAXLEN = 36;  MAXWID = 78;
len = 5;  wid = 21;

whos global
  Name          Size                    Bytes  Class

  MAXLEN        1x1                         8  double array (global)
  MAXWID        1x1                         8  double array (global)

Grand total is 2 elements using 16 bytes
```

Suggestions for Using Global Variables. A certain amount of risk is associated with using global variables and, because of this, it is recommended that you use them sparingly. You might, for example, unintentionally give a global variable in one function a name that is already used for a global variable in another function. When you run your application, one function may overwrite the variable used by the other. This error can be difficult to track down.

Another problem comes when you want to change the variable name. To make a change without introducing an error into the application, you must find every occurrence of that name in your code (and other people's code, if you share functions).

Alternatives to Using Global Variables. Instead of using a global variable, you may be able to

- Pass the variable to other functions as an additional argument. In this way, you make sure that any shared access to the variable is intentional.

  If this means that you have to pass a number of additional variables, you can put them into a structure or cell array and just pass it as one additional argument.

- Use a persistent variable (described in the next section), if you only need to make the variable persist in memory from one function call to the next.

Persistent Variables

Characteristics of persistent variables are

- You can declare and use them within M-file functions only.
- Only the function in which the variables are declared is allowed access to it.
- MATLAB does not clear them from memory when the function exits, so their value is retained from one function call to the next.

You must declare persistent variables before you can use them in a function. It is usually best to put your persistent declarations toward the beginning of the function. You would declare persistent variable `SUM_X` as follows:

```
persistent SUM_X
```

If you clear a function that defines a persistent variable (i.e., using `clear` *`functionname`* or `clear all`), or if you edit the M-file for that function, MATLAB clears all persistent variables used in that function.

You can use the `mlock` function to keep an M-file from being cleared from memory, thus keeping persistent variables in the M-file from being cleared as well.

Initializing Persistent Variables. When you declare a persistent variable, MATLAB initializes its value to an empty matrix, `[]`. After the declaration statement, you can assign your own value to it. This is often done using an `isempty` statement, as shown here:

```
function findSum(inputvalue)
persistent SUM_X

if isempty(SUM_X)
   SUM_X = 0;
end
SUM_X = SUM_X + inputvalue
```

This initializes the variable to 0 the first time you execute the function, and then it accumulates the value on each iteration.

Naming Variables

MATLAB variable names must begin with a letter, which may be followed by any combination of letters, digits, and underscores. MATLAB distinguishes between uppercase and lowercase characters, so `A` and `a` are not the same variable.

Although variable names can be of any length, MATLAB uses only the first `N` characters of the name, (where `N` is the number returned by the function `namelengthmax`), and ignores the rest. Hence, it is important to make each variable name unique in the first `N` characters to enable MATLAB to distinguish variables.

```
N = namelengthmax
N =
    63
```

The `genvarname` function can be useful in creating variable names that are both valid and unique. See the `genvarname` reference page to find out how to do this.

Verifying a Variable Name

You can use the `isvarname` function to make sure a name is valid before you use it. `isvarname` returns 1 if the name is valid, and 0 otherwise.

```
isvarname 8th_column
ans =
     0                    % Not valid - begins with a number
```

Avoid Using Function Names for Variables

When naming a variable, make sure you are not using a name that is already used as a function name, either one of your own M-file functions or one of the functions in the MATLAB language. If you define a variable with a function name, you will not be able to call that function until you either remove the variable from memory with the `clear` function, or invoke the function using `builtin`.

For example, if you enter the following command, you will not be able to use the MATLAB `disp` function until you clear the variable with `clear disp`.

```
disp = 50;
```

To test whether a proposed variable name is already used as a function name, use

```
which -all variable_name
```

Potential Conflict with Function Names

There are some MATLAB functions that have names that are commonly used as variable names in programming code. A few examples of such functions are `i`, `j`, `mode`, `char`, `size`, and `path`.

If you need to use a variable that is also the name of a MATLAB function, and have determined that you have no need to call the function, you should be aware that there is still a possibility for conflict. See the following two examples:

- "Variables Loaded From a MAT-File" on page 2-15
- "Variables In Evaluation Statements" on page 2-16

Variables Loaded From a MAT-File. The function shown below loads previously saved data from MAT-file `settings.mat`. It is supposed to display the value of one of the loaded variables, `mode`. However, `mode` is also the name of a MATLAB function and, in this case, MATLAB interprets it as the function and not the variable loaded from the MAT-file:

```
function show_mode
load settings;
whos mode
fprintf('Mode is set to %s\n', mode)
```

Assume that `mode` already exists in the MAT-file. Execution of the function shows that, even though `mode` is successfully loaded into the function workspace as a variable, when MATLAB attempts to operate on it in the last line, it interprets `mode` as a function. This results in an error:

```
show_mode
  Name       Size                      Bytes  Class

  mode       1x6                          12  char array

Grand total is 6 elements using 12 bytes

??? Error using ==> mode
Not enough input arguments.

Error in ==> show_mode at 4
fprintf('Mode is set to %s\n', mode)
```

Because MATLAB parses function M-files before they are run, it needs to determine before runtime which identifiers in the code are variables and which are functions. The function in this example does not establish `mode` as a variable name and, as a result, MATLAB interprets it as a function name instead.

There are several ways to make this function work as intended without having to change the variable name. Both indicate to MATLAB that the name represents a variable, and not a function:

- Name the variable explicitly in the `load` statement:

```
function show_mode
load settings mode;
whos mode
fprintf('Mode is set to %s\n', mode)
```

- Initialize the variable (e.g., set it to an empty matrix or empty string) at the start of the function:

```
function show_mode
mode = '';
load settings;
whos mode
fprintf('Mode is set to %s\n', mode)
```

Variables In Evaluation Statements. Variables used in evaluation statements such as `eval`, `evalc`, and `evalin` can also be mistaken for function names. The following M-file defines a variable named `length` that conflicts with MATLAB `length` function:

```
function find_area
eval('length = 12;  width = 36;');
fprintf('The area is %d\n', length .* width)
```

The second line of this code would seem to establish `length` as a variable name that would be valid when used in the statement on the third line. However, when MATLAB parses this line, it does not consider the *contents* of the string that is to be evaluated. As a result, MATLAB has no way of knowing that `length` was meant to be used as a variable name in this program, and the name defaults to a function name instead, yielding the following error:

```
find_area
??? Error using ==> length
Not enough input arguments.
```

To force MATLAB to interpret `length` as a variable name, use it in an explicit assignment statement first:

```
function find_area
length = [];
eval('length = 12;  width = 36;');
fprintf('The area is %d\n', length .* width)
```

Guidelines to Using Variables

The same guidelines that apply to MATLAB variables at the command line also apply to variables in M-files:

- You do not need to type or declare variables used in M-files (with the possible exception of designating them as `global` or `persistent`).
- Before assigning one variable to another, you must be sure that the variable on the right-hand side of the assignment has a value.
- Any operation that assigns a value to a variable creates the variable, if needed, or overwrites its current value, if it already exists.

Scope of a Variable

MATLAB stores variables in a part of memory called a workspace. The *base workspace* holds variables created during your interactive MATLAB session and also any variables created by running M-file scripts. Variables created at the MATLAB command prompt can also be used by scripts without having to declare them as global.

Functions do not use the base workspace. Every function has its own *function workspace*. Each function workspace is kept separate from the base workspace and all other workspaces to protect the integrity of the data used by that function. Even subfunctions that are defined in the same M-file have a separate function workspace.

Extending Variable Scope

In most cases, variables created within a function are known only within that function. These variables are not available at the MATLAB command prompt or to any other function or subfunction.

Passing Variables from Another Workspace. The most secure way to extend the scope of a function variable is to pass it to other functions as an argument in the function call. Since MATLAB passes data only by value, you also need to add the variable to the return values of any function that modifies its value.

Evaluating in Another Workspace Using evalin. Functions can also obtain variables from either the base or the caller's workspace using the `evalin` function. The example below, `compareAB_1`, evaluates a command in the context of the MATLAB command line, taking the values of variables `A` and `B` from the base workspace.

Define `A` and `B` in the base workspace:

```
A = [13 25 82 68 9 15 77];  B = [63 21 71 42 30 15 22];
```

Use `evalin` to evaluate the command `A(find(A<=B))` in the context of the MATLAB base workspace:

```
function C = compareAB_1
C = evalin('base', 'A(find(A<=B))');
```

Call the function. You do not have to pass the variables because they are made available to the function via the `evalin` function:

```
C = compareAB_1
C =
    13     9    15
```

You can also evaluate in the context of the caller's workspace by specifying `'caller'` (instead of `'base'`) as the first input argument to `evalin`.

Using Global Variables. A third way to extend variable scope is to declare the variable as `global` within every function that needs access to it. If you do this, you need make sure that no functions with access to the variable overwrite its value unintentionally. For this reason, it is recommended that you limit the use of global variables.

Create global vectors `A` and `B` in the base workspace:

```
global A
global B
A = [13 25 82 68 9 15 77];  B = [63 21 71 42 30 15 22];
```

Also declare them in the function to be called:

```
function C = compareAB_2
global A
```

```
global B

C = A(find(A<=B));
```

Call the function. Again, you do not have to pass A and B as arguments to the called function:

```
C = compareAB_2
C =
    13     9    15
```

Scope in Nested Functions

Variables within nested functions are accessible to more than just their immediate function. As a general rule, the scope of a local variable is the largest containing function body in which the variable appears, and all functions nested within that function. For more information on nested functions, see "Variable Scope in Nested Functions" on page 4-19.

Lifetime of a Variable

Variables created at the MATLAB command prompt or in an M-file script exist until you clear them or end your MATLAB session. Variables in functions exist only until the function completes unless they have been declared as global or persistent.

Keywords

The MATLAB software reserves certain words for its own use as keywords of the language. To list the keywords, type

```
iskeyword
ans =
    'break'
    'case'
    'catch'
       .
       .
       .
```

See the online function reference pages to learn how to use these keywords.

You should not use MATLAB keywords other than for their intended purpose. For example, a keyword should not be used as follows:

```
while = 5;
??? while = 5;
          |
Error: Expected a variable, function, or constant, found "=".
```

Special Values

Several functions return important special values that you can use in your M-files.

Function	Return Value
ans	Most recent answer (variable). If you do not assign an output variable to an expression, MATLAB automatically stores the result in ans.
eps	Floating-point relative accuracy. This is the tolerance the MATLAB software uses in its calculations.
intmax	Largest 8-, 16-, 32-, or 64-bit integer your computer can represent.
intmin	Smallest 8-, 16-, 32-, or 64-bit integer your computer can represent.
realmax	Largest floating-point number your computer can represent.
realmin	Smallest positive floating-point number your computer can represent.
pi	3.1415926535897...
i, j	Imaginary unit.
inf	Infinity. Calculations like n/0, where n is any nonzero real value, result in inf.
NaN	Not a Number, an invalid numeric value. Expressions like 0/0 and inf/inf result in a NaN, as do arithmetic operations involving a NaN. Also, if n is complex with a zero real part, then n/0 returns a value with a NaN real part.
computer	Computer type.
version	MATLAB version string.

Here are some examples that use these values in MATLAB expressions.

```
x = 2 * pi
x =
   6.2832

A = [3+2i 7-8i]
A =
   3.0000 + 2.0000i   7.0000 - 8.0000i

tol = 3 * eps
tol =
   6.6613e-016

intmax('uint64')
ans =
   18446744073709551615
```

Operators

In this section...

Arithmetic Operators

Arithmetic operators perform numeric computations, for example, adding two numbers or raising the elements of an array to a given power. The following table provides a summary. For more information, see the arithmetic operators reference page.

Operator	Description
`+`	Addition
`-`	Subtraction
`.*`	Multiplication
`./`	Right division
`.\`	Left division
`+`	Unary plus
`-`	Unary minus
`:`	Colon operator
`.^`	Power
`.'`	Transpose
`'`	Complex conjugate transpose
`*`	Matrix multiplication
`/`	Matrix right division
`\`	Matrix left division
`^`	Matrix power

Arithmetic Operators and Arrays

Except for some matrix operators, MATLAB arithmetic operators work on corresponding elements of arrays with equal dimensions. For vectors and rectangular arrays, both operands must be the same size unless one is a scalar. If one operand is a scalar and the other is not, MATLAB applies the scalar to every element of the other operand—this property is known as *scalar expansion*.

This example uses scalar expansion to compute the product of a scalar operand and a matrix.

```
A = magic(3)
A =
     8     1     6
     3     5     7
     4     9     2

3 * A
ans =
    24     3    18
     9    15    21
    12    27     6
```

Relational Operators

Relational operators compare operands quantitatively, using operators like “less than” and “not equal to.” The following table provides a summary. For more information, see the relational operators reference page.

Operator	Description
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
~=	Not equal to

Relational Operators and Arrays

The MATLAB relational operators compare corresponding elements of arrays with equal dimensions. Relational operators always operate element-by-element. In this example, the resulting matrix shows where an element of `A` is equal to the corresponding element of `B`.

```
A = [2 7 6;9 0 5;3 0.5 6];
B = [8 7 0;3 2 5;4 -1 7];

A == B
ans =
     0     1     0
     0     0     1
     0     0     0
```

For vectors and rectangular arrays, both operands must be the same size unless one is a scalar. For the case where one operand is a scalar and the other is not, MATLAB tests the scalar against every element of the other operand. Locations where the specified relation is true receive logical `1`. Locations where the relation is false receive logical `0`.

Relational Operators and Empty Arrays

The relational operators work with arrays for which any dimension has size zero, as long as both arrays are the same size or one is a scalar. However, expressions such as

```
A == []
```

return an error if A is not 0-by-0 or 1-by-1. This behavior is consistent with that of all other binary operators, such as `+`, `-`, `>`, `<`, `&`, `|`, etc.

To test for empty arrays, use the function

```
isempty(A)
```

Logical Operators

MATLAB offers three types of logical operators and functions:

- Element-wise — operate on corresponding elements of logical arrays.

- Bit-wise — operate on corresponding bits of integer values or arrays.
- Short-circuit — operate on scalar, logical expressions.

The values returned by MATLAB logical operators and functions, with the exception of bit-wise functions, are of type `logical` and are suitable for use with logical indexing.

Element-Wise Operators and Functions

The following logical operators and functions perform elementwise logical operations on their inputs to produce a like-sized output array.

The examples shown in the following table use vector inputs `A` and `B`, where

```
A = [0 1 1 0 1];
B = [1 1 0 0 1];
```

Operator	Description	Example
`&`	Returns `1` for every element location that is `true` (nonzero) in both arrays, and `0` for all other elements.	`A & B = 01001`
`|`	Returns `1` for every element location that is `true` (nonzero) in either one or the other, or both arrays, and `0` for all other elements.	`A | B = 11101`
`~`	Complements each element of the input array, `A`.	`~A = 10010`
`xor`	Returns `1` for every element location that is `true` (nonzero) in only one array, and `0` for all other elements.	`xor(A,B) = 10100`

For operators and functions that take two array operands, (`&`, `|`, and `xor`), both arrays must have equal dimensions, with each dimension being the same size. The one exception to this is where one operand is a scalar and the other is not. In this case, MATLAB tests the scalar against every element of the other operand.

Note MATLAB converts any finite nonzero, numeric values used as inputs to logical expressions to logical `1`, or `true`.

Operator Overloading. You can overload the `&`, `|`, and `~` operators to make their behavior dependent upon the class on which they are being used. Each of these operators has a representative function that is called whenever that operator is used. These are shown in the table below.

Logical Operation	Equivalent Function
`A & B`	`and(A, B)`
`A	B`	`or(A, B)`
`~A`	`not(A)`

Other Array Functions. Two other MATLAB functions that operate logically on arrays, but not in an elementwise fashion, are `any` and `all`. These functions show whether *any* or *all* elements of a vector, or a vector within a matrix or an array, are nonzero.

When used on a matrix, `any` and `all` operate on the columns of the matrix. When used on an N-dimensional array, they operate on the first nonsingleton dimension of the array. Or, you can specify an additional `dimension` input to operate on a specific dimension of the array.

The examples shown in the following table use array input `A`, where

```
A = [0   1   2;
     0  -3   8;
     0   5   0];
```

Function	Description	Example
`any(A)`	Returns `1` for a vector where *any* element of the vector is `true` (nonzero), and `0` if no elements are `true`.	`any(A) ans = 0 1 1`
`all(A)`	Returns `1` for a vector where *all* elements of the vector are `true` (nonzero), and `0` if all elements are not `true`.	`all(A) ans = 0 1 0`

Note The `all` and `any` functions ignore any `NaN` values in the input arrays.

Short-Circuiting in Elementwise Operators. When used in the context of an `if` or `while` expression, and only in this context, the elementwise `|` and `&` operators use short-circuiting in evaluating their expressions. That is, `A|B` and `A&B` ignore the second operand, `B`, if the first operand, `A`, is sufficient to determine the result.

So, although the statement `1|[]` evaluates to `false`, the same statement evaluates to `true` when used in either an `if` or `while` expression:

```
A = 1;   B = [];
if(A|B) disp 'The statement is true',  end;
   The statement is true
```

while the reverse logical expression, which does not short-circuit, evaluates to `false`

```
if(B|A) disp 'The statement is true',  end;
```

Another example of short-circuiting with elementwise operators shows that a logical expression such as the following, which under most circumstances is invalid due to a size mismatch between `A` and `B`,

```
A = [1 1];   B = [2 0 1];
A|B        % This generates an error.
```

works within the context of an `if` or `while` expression:

```
if (A|B) disp 'The statement is true',  end;
```

```
The statement is true
```

Logical Expressions Using the find Function. The `find` function determines the indices of array elements that meet a given logical condition. The function is useful for creating masks and index matrices. In its most general form, `find` returns a single vector of indices. This vector can be used to index into arrays of any size or shape.

For example,

```
A = magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

i = find(A > 8);
A(i) = 100
A =
   100     2     3   100
     5   100   100     8
   100     7     6   100
     4   100   100     1
```

Note An alternative to using `find` in this context is to index into the matrix using the logical expression itself. See the example below.

The last two statements of the previous example can be replaced with this one statement:

```
A(A > 8) = 100;
```

You can also use `find` to obtain both the row and column indices of a rectangular matrix for the array values that meet the logical condition:

```
A = magic(4)
A =
    16     2     3    13
```

```
     5    11    10     8
     9     7     6    12
     4    14    15     1

[row, col] = find(A > 12)
row =
     1
     4
     4
     1
col =
     1
     2
     3
     4
```

Bit-Wise Functions

The following functions perform bit-wise logical operations on nonnegative integer inputs. Inputs may be scalar or in arrays. If in arrays, these functions produce a like-sized output array.

The examples shown in the following table use scalar inputs A and B, where

```
A = 28;                  % binary 11100
B = 21;                  % binary 10101
```

Function	Description	Example
bitand	Returns the bit-wise AND of two nonnegative integer arguments.	bitand(A,B) = 20 (binary 10100)
bitor	Returns the bit-wise OR of two nonnegative integer arguments.	bitor(A,B) = 29 (binary 11101)

Function	Description	Example
`bitcmp`	Returns the bit-wise complement as an `n`-bit number, where `n` is the second input argument to `bitcmp`.	`bitcmp(A,5) = 3 (binary 00011)`
`bitxor`	Returns the bit-wise exclusive OR of two nonnegative integer arguments.	`bitxor(A,B) = 9 (binary 01001)`

Short-Circuit Operators

The following operators perform AND and OR operations on logical expressions containing scalar values. They are *short-circuit* operators in that they evaluate their second operand only when the result is not fully determined by the first operand.

Operator	Description
`&&`	Returns logical `1` (`true`) if both inputs evaluate to `true`, and logical `0` (`false`) if they do not.
`||`	Returns logical `1` (`true`) if either input, or both, evaluate to `true`, and logical `0` (`false`) if they do not.

The statement shown here performs an AND of two logical terms, `A` and `B`:

```
A && B
```

If `A` equals zero, then the entire expression will evaluate to logical `0` (`false`), regardless of the value of `B`. Under these circumstances, there is no need to evaluate `B` because the result is already known. In this case, MATLAB short-circuits the statement by evaluating only the first term.

A similar case is when you OR two terms and the first term is `true`. Again, regardless of the value of `B`, the statement will evaluate to `true`. There is no need to evaluate the second term, and MATLAB does not do so.

Advantage of Short-Circuiting. You can use the short-circuit operators to evaluate an expression only when certain conditions are satisfied. For example, you want to execute an M-file function only if the M-file resides on the current MATLAB path.

Short-circuiting keeps the following code from generating an error when the file, `myfun.m`, cannot be found:

```
comp = (exist('myfun.m') == 2) && (myfun(x) >= y)
```

Similarly, this statement avoids divide-by-zero errors when `b` equals zero:

```
x = (b ~= 0) && (a/b > 18.5)
```

You can also use the `&&` and `||` operators in `if` and `while` statements to take advantage of their short-circuiting behavior:

```
if (nargin >= 3) && (ischar(varargin{3}))
```

Operator Precedence

You can build expressions that use any combination of arithmetic, relational, and logical operators. Precedence levels determine the order in which MATLAB evaluates an expression. Within each precedence level, operators have equal precedence and are evaluated from left to right. The precedence rules for MATLAB operators are shown in this list, ordered from highest precedence level to lowest precedence level:

1. Parentheses `()`

2. Transpose `(.')`, power `(.^)`, complex conjugate transpose `(')`, matrix power `(^)`

3. Unary plus `(+)`, unary minus `(-)`, logical negation `(~)`

4. Multiplication `(.*)`, right division `(./)`, left division `(.\)`, matrix multiplication `(*)`, matrix right division `(/)`, matrix left division `(\)`

5. Addition `(+)`, subtraction `(-)`

6. Colon operator `(:)`

7. Less than (`<`), less than or equal to (`<=`), greater than (`>`), greater than or equal to (`>=`), equal to (`==`), not equal to (`~=`)
8. Element-wise AND (`&`)
9. Element-wise OR (`|`)
10. Short-circuit AND (`&&`)
11. Short-circuit OR (`||`)

Precedence of AND and OR Operators

MATLAB always gives the `&` operator precedence over the `|` operator. Although MATLAB typically evaluates expressions from left to right, the expression `a|b&c` is evaluated as `a|(b&c)`. It is a good idea to use parentheses to explicitly specify the intended precedence of statements containing combinations of `&` and `|`.

The same precedence rule holds true for the `&&` and `||` operators.

Overriding Default Precedence

The default precedence can be overridden using parentheses, as shown in this example:

```
A = [3 9 5];
B = [2 1 5];
C = A./B.^2
C =
    0.7500    9.0000    0.2000

C = (A./B).^2
C =
    2.2500   81.0000    1.0000
```

Comma-Separated Lists

In this section...

What Is a Comma-Separated List?

Typing in a series of numbers separated by commas gives you what is called a *comma-separated list*. The MATLAB software returns each value individually:

```
1, 2, 3
ans =
     1
ans =
     2
ans =
     3
```

Such a list, by itself, is not very useful. But when used with large and more complex data structures like MATLAB structures and cell arrays, the comma-separated list can enable you to simplify your MATLAB code.

Generating a Comma-Separated List

This section describes how to generate a comma-separated list from either a cell array or a MATLAB structure.

Generating a List from a Cell Array

Extracting multiple elements from a cell array yields a comma-separated list. Given a 4-by-6 cell array as shown here

```
C = cell(4, 6);
for k = 1:24,   C{k} = k * 2;   end

C
C =
    [2]    [10]    [18]    [26]    [34]    [42]
    [4]    [12]    [20]    [28]    [36]    [44]
    [6]    [14]    [22]    [30]    [38]    [46]
    [8]    [16]    [24]    [32]    [40]    [48]
```

extracting the fifth column generates the following comma-separated list:

```
C{:, 5}
ans =
    34
ans =
    36
ans =
    38
ans =
    40
```

This is the same as explicitly typing

```
C{1, 5}, C{2, 5}, C{3, 5}, C{4, 5}
```

Generating a List from a Structure

For structures, extracting a field of the structure that exists across one of its dimensions yields a comma-separated list.

Start by converting the cell array used above into a 4-by-1 MATLAB structure with six fields: `f1` through `f6`. Read field `f5` for all rows and MATLAB returns a comma-separated list:

```
S = cell2struct(C, {'f1', 'f2', 'f3', 'f4', 'f5', 'f6'}, 2);

S.f5
ans =
    34
ans =
```

```
    36
ans =
    38
ans =
    40
```

This is the same as explicitly typing

```
S(1).f5, S(2).f5, S(3).f5, S(4).f5
```

Assigning Output from a Comma-Separated List

You can assign any or all consecutive elements of a comma-separated list to variables with a simple assignment statement. Using the cell array `C` from the previous section, assign the first row to variables `c1` through `c6`:

```
C = cell(4, 6);
for k = 1:24,   C{k} = k * 2;   end

[c1 c2 c3 c4 c5 c6] = C{1,1:6};

c5
c5 =
    34
```

If you specify fewer output variables than the number of outputs returned by the expression, MATLAB assigns the first N outputs to those N variables, and then discards any remaining outputs. In this next example, MATLAB assigns `C{1,1:3}` to the variables `c1`, `c2`, and `c3`, and then discards `C{1,4:6}`:

```
[c1 c2 c3] = C{1,1:6};
```

You can assign structure outputs in the same manner:

```
S = cell2struct(C, {'f1', 'f2', 'f3', 'f4', 'f5', 'f6'}, 2);

[sf1 sf2 sf3] = S.f5;

sf3
sf3 =
    38
```

You also can use the `deal` function for this purpose.

Assigning to a Comma-Separated List

The simplest way to assign multiple values to a comma-separated list is to use the `deal` function. This function distributes all of its input arguments to the elements of a comma-separated list.

This example initializes a comma-separated list to a set of vectors in a cell array, and then uses `deal` to overwrite each element in the list:

```
c{1} = [31 07];    c{2} = [03 78];

c{:}
ans =
    31     7
ans =
     3    78


[c{:}] = deal([10 20],[14 12]);

c{:}
ans =
    10    20
ans =
    14    12
```

This example does the same as the one above, but with a comma-separated list of vectors in a structure field:

```
s(1).field1 = [31 07];    s(2).field1 = [03 78];

s.field1
ans =
    31     7
ans =
     3    78
```

```
[s.field1] = deal([10 20],[14 12]);

s.field1
ans =
    10    20
ans =
    14    12
```

How to Use the Comma-Separated Lists

Common uses for comma-separated lists are

- "Constructing Arrays" on page 2-38
- "Displaying Arrays" on page 2-39
- "Concatenation" on page 2-39
- "Function Call Arguments" on page 2-39
- "Function Return Values" on page 2-40

The following sections provide examples of using comma-separated lists with cell arrays. Each of these examples applies to MATLAB structures as well.

Constructing Arrays

You can use a comma-separated list to enter a series of elements when constructing a matrix or array. Note what happens when you insert a *list* of elements as opposed to adding the cell itself.

When you specify a list of elements with `C{:, 5}`, MATLAB inserts the four individual elements:

```
A = {'Hello', C{:, 5}, magic(4)}
A =
    'Hello'    [34]    [36]    [38]    [40]    [4x4 double]
```

When you specify the `C` cell itself, MATLAB inserts the entire cell array:

```
A = {'Hello', C, magic(4)}
A =
    'Hello'    {4x6 cell}    [4x4 double]
```

Displaying Arrays

Use a list to display all or part of a structure or cell array:

```
A{:}
ans =
    Hello
ans =
    34
ans =
    36
ans =
    38
     .
     .
     .
```

Concatenation

Putting a comma-separated list inside square brackets extracts the specified elements from the list and concatenates them:

```
A = [C{:, 5:6}]
A =
    34    36    38    40    42    44    46    48

whos A
  Name      Size                   Bytes  Class

  A         1x8                       64  double array
```

Function Call Arguments

When writing the code for a function call, you enter the input arguments as a list with each argument separated by a comma. If you have these arguments stored in a structure or cell array, then you can generate all or part of the argument list from the structure or cell array instead. This can be especially useful when passing in variable numbers of arguments.

This example passes several attribute-value arguments to the `plot` function:

```
X = -pi:pi/10:pi;
Y = tan(sin(X)) - sin(tan(X));

C{1,1} = 'LineWidth';         C{2,1} = 2;
C{1,2} = 'MarkerEdgeColor';   C{2,2} = 'k';
C{1,3} = 'MarkerFaceColor';   C{2,3} = 'g';

plot(X, Y, '--rs', C{:})
```

Function Return Values

MATLAB functions can also return more than one value to the caller. These values are returned in a list with each value separated by a comma. Instead of listing each return value, you can use a comma-separated list with a structure or cell array. This becomes more useful for those functions that have variable numbers of return values.

This example returns four values to a cell array:

```
C = cell(1, 4);
[C{:}] = fileparts('work/mytests/strArrays.mat')
C =
    'work/mytests'    'strArrays'    '.mat'     ''
```

Fast Fourier Transform Example

The `fftshift` function swaps the left and right halves of each dimension of an array. For a simple vector such as `[0 2 4 6 8 10]` the output would be `[6 8 10 0 2 4]`. For a multidimensional array, `fftshift` performs this swap along each dimension.

`fftshift` uses vectors of indices to perform the swap. For the vector shown above, the index `[1 2 3 4 5 6]` is rearranged to form a new index `[4 5 6 1 2 3]`. The function then uses this index vector to reposition the elements. For a multidimensional array, `fftshift` must construct an index vector for each dimension. A comma-separated list makes this task much simpler.

Here is the `fftshift` function:

```
function y = fftshift(x)
```

```
numDims = ndims(x);
idx = cell(1, numDims);

for k = 1:numDims
   m = size(x, k);
   p = ceil(m/2);
   idx{k} = [p+1:m 1:p];
   end

y = x(idx{:});
```

The function stores the index vectors in cell array `idx`. Building this cell array is relatively simple. For each of the `N` dimensions, determine the size of that dimension and find the integer index nearest the midpoint. Then, construct a vector that swaps the two halves of that dimension.

By using a cell array to store the index vectors and a comma-separated list for the indexing operation, `fftshift` shifts arrays of any dimension using just a single operation: `y = x(idx{:})`. If you were to use explicit indexing, you would need to write one `if` statement for each dimension you want the function to handle:

```
if ndims(x) == 1
   y = x(index1);
else if ndims(x) == 2
   y = x(index1, index2);
end
```

Another way to handle this without a comma-separated list would be to loop over each dimension, converting one dimension at a time and moving data each time. With a comma-separated list, you move the data just once. A comma-separated list makes it very easy to generalize the swapping operation to an arbitrary number of dimensions.

Program Control Statements

In this section...

Conditional Control — if, switch

This group of control statements enables you to select at run-time which block of code is executed. To make this selection based on whether a condition is `true` or `false`, use the `if` statement (which may include `else` or `elseif`). To select from a number of possible options depending on the value of an expression, use the `switch` and `case` statements (which may include `otherwise`).

You cannot define nested functions within a conditional control block. Nested functions must always be defined at the top level of a function.

if, else, and elseif

`if` evaluates a logical expression and executes a group of statements based on the value of the expression. In its simplest form, its syntax is

```
if logical_expression
    statements
end
```

If the logical expression is `true` (that is, if it evaluates to logical `1`), the MATLAB software executes all the statements between the `if` and `end` lines. It resumes execution at the line following the `end` statement. If the condition is `false` (evaluates to logical `0`), MATLAB skips all the statements between the `if` and `end` lines, and resumes execution at the line following the `end` statement.

For example,

```
if rem(a, 2) == 0
    disp('a is even')
    b = a/2;
end
```

You can nest any number of `if` statements.

If the logical expression evaluates to a nonscalar value, all the elements of the argument must be nonzero. For example, assume `X` is a matrix. Then the statement

```
if X
    statements
end
```

is equivalent to

```
if all(X(:))
    statements
end
```

The `else` and `elseif` statements further conditionalize the `if` statement:

- The `else` statement has no logical condition. The statements associated with it execute if the preceding `if` (and possibly `elseif` condition) evaluates to logical `0` (`false`).
- The `elseif` statement has a logical condition that it evaluates if the preceding `if` (and possibly `elseif` condition) is `false`. The statements associated with it execute if its logical condition evaluates to logical `1` (`true`). You can have multiple `elseif` statements within an `if` block.

  ```
  if n < 0            % If n negative, display error message.
      disp('Input must be positive');
  elseif rem(n,2) == 0 % If n positive and even, divide by 2.
      A = n/2;
  else
      A = (n+1)/2;     % If n positive and odd, increment and divide.
  end
  ```

if Statements and Empty Arrays. An `if` condition that reduces to an empty array represents a `false` condition. That is,

```
if A
   S1
else
   S0
end
```

executes statement `S0` when `A` is an empty array.

switch, case, and otherwise

`switch` executes certain statements based on the value of a variable or expression. Its basic form is

```
switch expression (scalar or string)
   case value1
      statements       % Executes if expression is value1
   case value2
      statements       % Executes if expression is value2
     .
     .
     .
   otherwise
      statements       % Executes if expression does not
                        % match any case
end
```

This block consists of

- The word `switch` followed by an expression to evaluate.
- Any number of `case` groups. These groups consist of the word `case` followed by a possible value for the expression, all on a single line. Subsequent lines contain the statements to execute for the given value of the expression. These can be any valid MATLAB statement including another `switch` block. Execution of a `case` group ends when MATLAB encounters the next `case` statement or the `otherwise` statement. Only the first matching `case` is executed.
- An optional `otherwise` group. This consists of the word `otherwise`, followed by the statements to execute if the expression's value is not handled by any of the preceding `case` groups. Execution of the `otherwise` group ends at the `end` statement.

- An end statement.

switch works by comparing the input expression to each case value. For numeric expressions, a case statement is true if (value==expression). For string expressions, a case statement is true if strcmp(value,expression).

The code below shows a simple example of the switch statement. It checks the variable input_num for certain values. If input_num is -1, 0, or 1, the case statements display the value as text. If input_num is none of these values, execution drops to the otherwise statement and the code displays the text 'other value'.

```
switch input_num
    case -1
        disp('negative one');
    case 0
        disp('zero');
    case 1
        disp('positive one');
    otherwise
         disp('other value');
end
```

Note For C programmers, unlike the C language switch construct, the MATLAB switch does not "fall through." That is, if the first case statement is true, other case statements do not execute. Therefore, break statements are not used.

---

switch can handle multiple conditions in a single case statement by enclosing the case expression in a cell array.

```
switch var
    case 1
        disp('1')
    case {2,3,4}
        disp('2 or 3 or 4')
    case 5
        disp('5')
    otherwise
```

```
        disp('something else')
end
```

Loop Control — for, while, continue, break

With loop control statements, you can repeatedly execute a block of code, looping back through the block while keeping track of each iteration with an incrementing index variable. Use the `for` statement to loop a specific number of times. The `while` statement is more suitable for basing the loop execution on how long a condition continues to be true or false. The `continue` and `break` statements give you more control on exiting the loop.

You cannot define nested functions within a loop control block. Nested functions must always be defined at the top level of a function.

Note You can often speed up the execution of MATLAB code by replacing `for` and `while` loops with vectorized code. See "Techniques for Improving Performance" on page 10-4 for more information on this.

---

for

The `for` loop executes a statement or group of statements a predetermined number of times. Its syntax is

```
for index = start:increment:end
    statements
end
```

The default increment is `1`. You can specify any increment, including a negative one. For positive indices, execution terminates when the value of the index exceeds the *end* value; for negative increments, it terminates when the index is less than the end value.

For example, this loop executes five times.

```
for n = 2:6
    x(n) = 2 * x(n - 1);
end
```

You can nest multiple `for` loops.

```
for m = 1:5
    for n = 1:100
        A(m, n) = 1/(m + n - 1);
    end
end
```

Note You can often speed up the execution of MATLAB code by replacing `for` and `while` loops with vectorized code. See "Vectorizing Loops" on page 10-4 for details.

Using Arrays as Indices. The index of a `for` loop can be an array. For example, consider an m-by-n array `A`. The statement

```
for k = A
    statements
end
```

sets `k` equal to the vector `A(:,i)`, where `i` is the iteration number of the loop. For the first loop iteration, `k` is equal to `A(:,1)`; for the second, `k` is equal to `A(:,2)`; and so on until `k` equals `A(:,n)`. That is, the loop iterates for a number of times equal to the number of columns in `A`. For each iteration, `k` is a vector containing one of the columns of `A`.

while

The `while` loop executes a statement or group of statements repeatedly as long as the controlling expression is true (`1`). Its syntax is

```
while expression
    statements
end
```

If the expression evaluates to a matrix, all its elements must be `1` for execution to continue. To reduce a matrix to a scalar value, use the `all` and `any` functions.

For example, this `while` loop finds the first integer `n` for which `n!` (`n` factorial) is a 100-digit number.

```
n = 1;
while prod(1:n) < 1e100
    n = n + 1;
end
```

Exit a `while` loop at any time using the `break` statement.

while Statements and Empty Arrays. A `while` condition that reduces to an empty array represents a `false` condition. That is,

```
while A, S1, end
```

never executes statement `S1` when `A` is an empty array.

continue

The `continue` statement passes control to the next iteration of the `for` or `while` loop in which it appears, skipping any remaining statements in the body of the loop. In `for` loops, the loop counter is incremented by the appropriate value (either 1 or the specified step value) at the start of the next iteration.

`continue` works the same way in nested loops. That is, execution continues at the beginning of the loop in which the `continue` statement was encountered.

The example below shows a `continue` loop that counts the lines of code in the file, `magic.m`, skipping all blank lines and comments. A `continue` statement is used to advance to the next line in `magic.m` without incrementing the count whenever a blank line or comment line is encountered.

```
fid = fopen('magic.m','r');
count = 0;
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line) || strncmp(line,'%',1) || ~ischar(line)
        continue
    end
    count = count + 1;
end
```

```
fprintf('%d lines\n',count);
fclose(fid);
```

break

The `break` statement terminates the execution of a `for` loop or `while` loop. When a `break` statement is encountered, execution continues with the next statement outside of the loop. In nested loops, `break` exits from the innermost loop only.

The example below shows a `while` loop that reads the contents of the file `fft.m` into a MATLAB character array. A `break` statement is used to exit the `while` loop when the first empty line is encountered. The resulting character array contains the M-file help for the `fft` program.

```
fid = fopen('fft.m','r');
s = '';

while ~feof(fid)
   line = fgetl(fid);
   if isempty(line) || ~ischar(line)
      break
   end
   s = sprintf('%s%s\n', s, line);
end
disp(s);

fclose(fid);
```

Error Control — try, catch

Error control statements provide a way for you to take certain actions in the event of an error. Use the `try` statement to test whether a certain command in your code generates an error. If an error does occur within the `try` block, MATLAB immediately jumps to the corresponding `catch` block. The `catch` part of the statement needs to respond in some way to the error.

You cannot define nested functions within a `try-catch` block. Nested functions must always be defined at the top level of a function.

try and catch

The general form of a try-catch statement sequence is

```
try
   statement
   ...
   statement
catch
   statement
   ...
   statement
end
```

In this sequence, the statements between `try` and `catch` are executed until an error occurs. The statements between `catch` and `end` are then executed. Use `lasterr` to see the cause of the error. If an error occurs between `catch` and `end`, MATLAB terminates execution unless another `try-catch` sequence has been established.

Program Termination — return

Program termination control enables you to exit from your program at some point prior to its normal termination point.

return

After a MATLAB function runs to completion, it terminates and returns control either to the function that called it, or to the keyboard. If you need to exit a function prior to the point of normal completion, you can force an early termination using the `return` function. `return` immediately terminates the current sequence of commands and exits the currently running function.

`return` is also used to terminate `keyboard` mode.

Dates and Times

In this section...

Overview

The MATLAB software represents date and time information in either of three formats: date strings, serial date numbers, or date vectors. You have the choice of using any of these formats. If you work with more than one date and time format, MATLAB provides functions to help you easily convert from one format to another, (e.g., from a string to a serial date number).

When using date strings, you have an additional option of choosing from 19 different string styles to express date and/or time information.

Types of Date Formats

The three MATLAB date and time formats are

- "Date Strings" on page 2-52
- "Serial Date Numbers" on page 2-52
- "Date Vectors" on page 2-53

This table shows examples of the three formats.

Date Format	Example
Date string	`02-Oct-1996`

Date Format	Example
Serial date number	729300
Date vector	1996 10 2 0 0 0

Date Strings

There are a number of different styles in which to express date and time information as a date string. For example, several possibilities for October 31, 2003 at 3:45:17 in the afternoon are

```
31-Oct-2003 15:45:17
10/31/03
15:45:17
03:45:17 PM
```

If you are working with a small number of dates at the MATLAB command line, then date strings are often the most convenient format to use.

Note The MATLAB `date` function returns the current date as a string.

Serial Date Numbers

A serial date number represents a calendar date as the number of days that has passed since a fixed base date. In MATLAB, serial date number 1 is January 1, 0000. MATLAB also uses serial time to represent fractions of days beginning at midnight; for example, 6 p.m. equals 0.75 serial days. So the string '31-Oct-2003, 6:00 pm' in MATLAB is date number 731885.75.

MATLAB works internally with serial date numbers. If you are using functions that handle large numbers of dates or doing extensive calculations with dates, you get better performance if you use date numbers.

Note The MATLAB `now` function returns the current date and time as a serial date number.

Date Vectors

Date vectors are an internal format for some MATLAB functions; you do not typically use them in calculations. A date vector contains the elements `[year month day hour minute second]`.

Note The MATLAB `clock` function returns the current date and time as a serial vector.

Conversions Between Date Formats

Functions that convert between date formats are shown below.

Function	Description
`datenum`	Convert a date string to a serial date number.
`datestr`	Convert a serial date number to a date string.
`datevec`	Split a date number or date string into individual date elements.

Here are some examples of conversions from one date format to another:

```
d1 = datenum('02-Oct-1996')
d1 =
    729300

d2 = datestr(d1 + 10)
d2 =
   12-Oct-1996

dv1 = datevec(d1)
dv1 =
    1996       10        2        0        0        0

dv2 = datevec(d2)
dv2 =
    1996       10       12        0        0        0
```

Date String Formats

The `datenum` function is important for doing date calculations efficiently. `datenum` takes an input string in any of several formats, with `'dd-mmm-yyyy'`, `'mm/dd/yyyy'`, or `'dd-mmm-yyyy, hh:mm:ss.ss'` most common. You can form up to six fields from letters and digits separated by any other characters:

- The day field is an integer from 1 to 31.
- The month field is either an integer from 1 to 12 or an alphabetic string with at least three characters.
- The year field is a nonnegative integer: if only two digits are specified, then a year 19yy is assumed; if the year is omitted, then the current year is used as a default.
- The hours, minutes, and seconds fields are optional. They are integers separated by colons or followed by `'AM'` or `'PM'`.

For example, if the current year is 1996, then these are all equivalent:

```
'17-May-1996'
'17-May-96'
'17-May'
'May 17, 1996'
'5/17/96'
'5/17'
```

and both of these represent the same time:

```
'17-May-1996, 18:30'
'5/17/96/6:30 pm'
```

Note that the default format for numbers-only input follows the American convention. Thus `3/6` is March 6, not June 3.

If you create a vector of input date strings, use a column vector and be sure all strings are the same length. Fill in with spaces or zeros.

Output Formats

The command `datestr(D, dateform)` converts a serial date `D` to one of 19 different date string output formats showing date, time, or both. The default output for dates is a day-month-year string: 01-Mar-1996. You select an alternative output format by using the optional integer argument `dateform`.

This table shows the date string formats that correspond to each `dateform` value.

dateform	Format	Description
0	`01-Mar-1996 15:45:17`	day-month-year hour:minute:second
1	`01-Mar-1996`	day-month-year
2	`03/01/96`	month/day/year
3	`Mar`	month, three letters
4	`M`	month, single letter
5	`3`	month
6	`03/01`	month/day
7	`1`	day of month
8	`Wed`	day of week, three letters
9	`W`	day of week, single letter
10	`1996`	year, four digits
11	`96`	year, two digits
12	`Mar96`	month year
13	`15:45:17`	hour:minute:second
14	`03:45:17 PM`	hour:minute:second AM or PM
15	`15:45`	hour:minute
16	`03:45 PM`	hour:minute AM or PM
17	`Q1-96`	calendar quarter-year
18	`Q1`	calendar quarter

Converting Output Format with datestr

Here are some examples of converting the date March 1, 1996 to various forms using the `datestr` function:

```
d = '01-Mar-1999'
d =
   01-Mar-1999

datestr(d)
ans =
     01-Mar-1999

datestr(d, 2)
ans =
     03/01/99

datestr(d, 17)
ans =
     Q1-99
```

Current Date and Time

The function `date` returns a string for today's date:

```
date
ans =
     02-Oct-1996
```

The function `now` returns the serial date number for the current date and time:

```
now
ans =
     729300.71

datestr(now)
ans =
     02-Oct-1996 16:56:16

datestr(floor(now))
ans =
     02-Oct-1996
```

Function Summary

MATLAB provides the following functions for time and date handling:

- Current Date and Time Functions on page 2-57
- Conversion Functions on page 2-57
- Utility Functions on page 2-57
- Timing Measurement Functions on page 2-58

Current Date and Time Functions

Function	Description
`clock`	Return the current date and time as a date vector.
`date`	Return the current date as date string.
`now`	Return the current date and time as serial date number.

Conversion Functions

Function	Description
`datenum`	Convert to a serial date number.
`datestr`	Convert to a string representation of the date.
`datevec`	Convert to a date vector.

Utility Functions

Function	Description
`addtodate`	Modify a date number by field.
`calendar`	Return a matrix representing a calendar.
`datetick`	Label axis tick lines with dates.
`eomday`	Return the last day of a year and month.
`weekday`	Return the current day of the week.

Timing Measurement Functions

Function	Description
`cputime`	Return the total CPU time used by MATLAB since it was started.
`etime`	Return the time elapsed between two date vectors.
`tic, toc`	Measure the time elapsed between invoking `tic` and `toc`.

Regular Expressions

In this section...

Overview

A regular expression is a string of characters that defines a certain pattern. You would normally use a regular expression in searching through text for a group of words that matches this pattern, perhaps while parsing program input, or while processing a block of text.

The string `'Joh?n\w*'` is an example of a regular expression. It defines a pattern that starts with the letters `Jo`, is optionally followed by the letter `h` (indicated by `'h?'`), is then followed by the letter `n`, and ends with any number of non-whitespace characters (indicated by `'\w*'`). This pattern matches any of the following:

```
Jon, John, Jonathan, Johnny
```

The MATLAB software supports most of the operators, or *metacharacters*, commonly used with regular expressions and provides several functions to use in searching and replacing text with these expressions.

MATLAB Regular Expression Functions

Several MATLAB functions support searching and replacing characters using regular expressions:

Function	Description
`regexp`	Match regular expression.
`regexpi`	Match regular expression, ignoring case.
`regexprep`	Replace string using regular expression.
`regexptranslate`	Translate string into regular expression.

See the function reference pages to obtain more information on these functions. For more information on how to use regular expressions in general, consult a reference on that subject.

The `regexp` and `regexpi` functions return up to seven outputs in the order shown in the reference page for `regexp`. You can select specific outputs to be returned by using one or more of the following qualifiers with these commands:

Qualifier	Value Returned
`'start'`	Starting index of each substring matching the expression
`'end'`	Ending index of each substring matching the expression
`'tokenExtents'`	Starting and ending indices of each substring matching a token in the expression
`'match'`	Text of each substring matching the expression
`'tokens'`	Text of each token captured

Qualifier	Value Returned
`'names'`	Name and text of each named token captured
`'split'`	Treating each match as a delimiter, the text of each substring between such delimiters.

There is an additional qualifier named `'once'` that you can use to return only the first match found.

Character Types

Tables and examples in this and other sections that follow show the operators and syntax supported by the `regexp`, `regexpi`, and `regexprep` functions in MATLAB. Expressions shown in the left column have special meaning and match one or more characters according to the usage described in the right column. Any character not having a special meaning, for example, any alphabetic character, matches that same character literally. To force one of the regular expression functions to interpret a sequence of characters literally (rather than as an operator) use the `regexptranslate` function.

Character types represent either a specific set of characters (e.g., uppercase) or a certain type of character (e.g., non-whitespace).

Operator	Usage
`.`	Any single character, including white space
$[c_1c_2c_3]$	Any character contained within the brackets: c_1 or c_2 or c_3
$[\hat{}c_1c_2c_3]$	Any character not contained within the brackets: anything but c_1 or c_2 or c_3
$[c_1\text{-}c_2]$	Any character in the range of c_1 through c_2
`\s`	Any white-space character; equivalent to `[ \f\n\r\t\v]`
`\S`	Any non-whitespace character; equivalent to `[^ \f\n\r\t\v]`

Operator	Usage
`\w`	Any alphabetic, numeric, or underscore character. For English character sets, this is equivalent to `[a-zA-Z_0-9]`.
`\W`	Any character that is not alphabetic, numeric, or underscore. For English character sets, this is equivalent to `[^a-zA-Z_0-9]`.
`\d`	Any numeric digit; equivalent to `[0-9]`
`\D`	Any nondigit character; equivalent to `[^0-9]`

The following examples demonstrate how to use the character classes listed above. See the `regexp` reference page for help with syntax. Most of these examples use the following string:

```
str = 'The rain in Spain falls mainly on the plain.';
```

Any Character — .

The `.` operator matches any single character, including white space.

Example 1 — Matching Any Character. Use the dot (`.`) operator to locate sequences of five consecutive characters that end with `'ain'`. The regular expression used in this example is

```
expr = '..ain';
```

Find each occurrence of the expression `expr` within the input string `str`. Return a vector of the indices at which any matches begin:

```
str = 'The rain in Spain falls mainly on the plain.';

startIndex = regexp(str, expr)
startIndex =
     4    13    24    39
```

Here is the input string with the returned `startIndex` values shown below it. Note that the dot operator not only matches the letters in the string, but white-space characters as well:

```
The rain in Spain falls mainly on the plain.
   |        |           |               |
   4       13          24              39
```

If you would prefer to have MATLAB return the text of the matching substrings, use the `'match'` qualifier in the command:

```
matchStr = regexp(str, expr, 'match')
matchStr =
    ' rain'    'Spain'    ' main'    'plain'
```

Example 2 — Returning Strings Rather than Indices. Here is the same example, this time specifying the command qualifier `'match'`. In this case, `regexp` returns the *text* of the matching strings rather than the starting index:

```
regexp(str, '..ain', 'match')
ans =
    ' rain'    'Spain'    ' main'    'plain'
```

Selected Characters — [c1c2c3]

Use `[`$c_1c_2c_3$`]` in an expression to match selected characters `r`, `p`, or `m` followed by `'ain'`. Specify two qualifiers this time, `'match'` and `'start'`, along with an output argument for each, `mat` and `idx`. This returns the matching strings and the starting indices of those strings:

```
[mat idx] = regexp(str, '[rpm]ain', 'match', 'start')
mat =
    'rain'    'pain'    'main'
idx =
     5    14    25
```

Range of Characters — [c1 - c2]

Use [c_1-c_2] in an expression to find words that begin with a letter in the range of A through Z:

```
[mat idx] = regexp(str, '[A-Z]\w*', 'match', 'start')
mat =
    'The'    'Spain'
idx =
     1    13
```

Word and White-Space Characters — \w, \s

Use \w and \s in an expression to find words that end with the letter n followed by a white-space character. Add a new qualifier, 'end', to return the str index that marks the end of each match:

```
[mat ix1 ix2] = regexp(str, '\w*n\s', 'match', 'start', 'end')
mat =
    'rain '    'in '    'Spain '    'on '
ix1 =
     5    10    13    32
ix2 =
     9    12    18    34
```

Numeric Digits — \d

Use \d to find numeric digits in the following string:

```
numstr = 'Easy as 1, 2, 3';

[mat idx] = regexp(numstr, '\d', 'match', 'start')
mat =
    '1'    '2'    '3'
idx =
     9    12    15
```

Character Representation

The following character combinations represent specific character and numeric values.

Operator	Usage
`\a`	Alarm (beep)
`\\`	Backslash
`\$`	Dollar sign
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\oN` or `\o{N}`	Character of octal value `N`
`\xN` or `\x{N}`	Character of hexadecimal value `N`
`\char`	If a character has special meaning in a regular expression, precede it with backslash (`\`) to match it literally.

Octal and Hexadecimal — \o, \x

Use `\x` and `\o` in an expression to find a comma (hex `2C`) followed by a space (octal `40`) followed by the character `2`:

```
numstr = 'Easy as 1, 2, 3';

[mat idx] = regexp(numstr, '\x2C\o{40}2', 'match', 'start')
mat =
    ', 2'
idx =
    10
```

Grouping Operators

When you need to use one of the regular expression operators on a number of consecutive elements in an expression, group these elements together with one of the grouping operators and apply the operation to the entire group. For example, this command matches a capital letter followed by a numeral and then an optional space character. These elements have to occur at least two times in succession for there to be a match. To apply the `{2,}` multiplier to all three consecutive characters, you can first make a group of the characters and then apply the `(?:)` quantifier to this group:

```
regexp('B5 A2 6F 63 R6 P4 B2 BC', '(?:[A-Z]\d\s?){2,}', 'match')
ans =
    'B5 A2 '    'R6 P4 B2 '
```

There are three types of explicit grouping operators that you can use when you need to apply an operation to more than just one element in an expression. Also in the grouping category is the alternative match (logical OR) operator, `|`. This creates two or more groups of elements in the expression and applies an operation to one of the groups.

Operator	Usage
`(expr)`	Group regular expressions and capture tokens.
`(?:expr)`	Group regular expressions, but do not capture tokens.
`(?>expr)`	Group atomically.
$expr_1 \| expr_2$	Match expression $expr_1$ or expression $expr_2$.

Grouping and Capture — (expr)

When you enclose an expression in parentheses, MATLAB not only treats all of the enclosed elements as a group, but also captures a token from these elements whenever a match with the input string is found. For an example of how to use this, see "Using Tokens — Example 1" on page 2-80.

Grouping Only — (?:expr)

Use `(?:expr)` to group a non-vowel (consonant, numeric, whitespace, punctuation, etc.) followed by a vowel in the palindrome `pstr`. Specify at least

two consecutive occurrences ({2,}) of this group. Return the starting and ending indices of the matched substrings:

```
pstr = 'Marge lets Norah see Sharon''s telegram';
expr = '(?:[^aeiou][aeiou]){2,}';

[mat ix1 ix2] = regexp(pstr, expr, 'match', 'start', 'end')
mat =
    'Nora'    'haro'    'tele'
ix1 =
    12    23    31
ix2 =
    15    26    34
```

Remove the grouping, and the {2,} now applies only to [aeiou]. The command is entirely different now as it looks for a non-vowel followed by at least two consecutive vowels:

```
expr = '[^aeiou][aeiou]{2,}';

[mat ix1 ix2] = regexp(pstr, expr, 'match', 'start', 'end')
mat =
    'see'
ix1 =
    18
ix2 =
    20
```

Alternative Match — expr1|expr2

Use $p_1|p_2$ to pick out words in the string that start with let or tel:

```
regexpi(pstr, '(let|tel)\w+', 'match')
ans =
    'lets'    'telegram'
```

Nonmatching Operators

The `comment` operator enables you to insert comments into your code to make it more maintainable. The text of the comment is ignored by MATLAB when matching against the input string.

Operator	Usage
`(?#comment)`	Insert a comment into the expression. Comments are ignored in matching.

Including Comments — (?#expr)

Use `(?#expr)` to add a comment to this expression that matches capitalized words in `pstr`. Comments are ignored in the process of finding a match:

```
regexp(pstr, '(?# Match words in caps)[A-Z]\w+', 'match')
ans =
    'Marge'    'Norah'    'Sharon'
```

Positional Operators

Positional operators in an expression match parts of the input string not by content, but by where they occur in the string (e.g., the first N characters in the string).

Operator	Usage
`^expr`	Match `expr` if it occurs at the beginning of the input string.
`expr$`	Match `expr` if it occurs at the end of the input string.
`\<expr`	Match `expr` when it occurs at the beginning of a word.
`expr\>`	Match `expr` when it occurs at the end of a word.
`\<expr\>`	Match `expr` when it represents the entire word.

Start and End of String Match — ^expr, expr$

Use ^expr to match words starting with the letter m or M only when it begins the string, and expr$ to match words ending with m or M only when it ends the string:

```
regexpi(pstr, '^m\w*|\w*m$', 'match')
ans =
    'Marge'    'telegram'
```

Start and End of Word Match — \<expr, expr\>

Use \<expr to match any words starting with n or N, or ending with e or E:

```
regexpi(pstr, '\<n\w*|\w*e\>', 'match')
ans =
    'Marge'    'Norah'    'see'
```

Exact Word Match — \<expr\>

Use \<expr\> to match a word starting with an n or N and ending with an h or H:

```
regexpi(pstr, '\<n\w*h\>', 'match')
ans =
    'Norah'
```

Lookaround Operators

Lookaround operators tell MATLAB to look either ahead or behind the current location in the string for a specified expression. If the expression is found, MATLAB attempts to match a given pattern.

This table shows the four lookaround expressions: lookahead, negative lookahead, lookbehind, and negative lookbehind.

Operator	Usage
(?=expr)	Look ahead from current position and test if expr is found.

Operator	Usage
`(?!expr)`	Look ahead from current position and test if `expr` is not found
`(?<=expr)`	Look behind from current position and test if `expr` is found.
`(?<!expr)`	Look behind from current position and test if `expr` is not found.

Lookaround operators do not change the current parsing location in the input string. They are more of a condition that must be satisfied for a match to occur.

For example, the following command uses an expression that matches alphabetic, numeric, or underscore characters (`\w*`) that meet the condition that they *look ahead to* (i.e., are immediately followed by) the letters `vision`. The resulting match includes only that part of the string that matches the `\w*` operator; it does not include those characters that match the lookahead expression `(?=vision)`:

```
[s e] = regexp('telegraph television telephone', ...
               '\w*(?=vision)', 'start', 'end')
s =
    11
e =
    14
```

If you repeat this command and match one character beyond the lookahead expression, you can see that parsing of the input string resumes at the letter `v`, thus demonstrating that matching the lookahead operator has not consumed any characters in the string:

```
regexp('telegraph television telephone', ...
       '\w*(?=vision).', 'match')
ans =
    'telev'
```

Note You can also use lookaround operators to perform a logical AND of two elements. See "Using Lookaround as a Logical Operator" on page 2-74.

Using the Lookahead Operator — expr(?=test)

Example 1 — Simple Lookahead Example. The first `regexp` statement below finds all 3-character sequences that end with the letters `ai`. The second statement, which uses lookahead operation, matches only single characters. The `(?=ai)` in the expression serves only as a condition for the match; it is not part of the match itself:

```
str = 'The rain in Spain falls mainly on the plain.';

% In this statement, 'ai' is part of the match.
regexp(str, '.ai', 'match')
ans =
    'rai'    'pai'    'mai'    'lai'

% In this statement, 'ai' is a condition for match.
regexp(str, '.(?=ai)', 'match')
ans =
    'r'    'p'    'm'    'l'
```

Repeat these two commands but, this time, also look for an additional character that follows the `ai` sequence. Note that, in the second `regexp` statement, parsing for the dot (`.`) that follows the `(?=ai)` lookahead begins immediately after the match for the first dot, and not after the `ai`, as it does in the first statement:

```
regexp(str, '.ai.', 'match')
ans =
    'rain'    'pain'    'main'    'lain'

regexp(str, '.(?=ai).', 'match')
ans =
    'ra'    'pa'    'ma'    'la'
```

Example 2 — Lookahead.

Look ahead to a file name (`fileread.m`), and return the name of the directory in which it resides:

```
str = which('fileread')
str =
```

```
   C:\Akernel\perfect\matlab\toolbox\matlab\iofun\fileread.m

regexp(str, '\w+(?=\\\w+\.[mp])', 'match')
ans =
    'iofun'
```

Using the Negative Lookahead Operator — expr(?!test)

Example — Negative Lookbehind and Lookahead. Generate a series of sequential numbers:

```
n = num2str(5:15)
n =
   5    6    7    8    9   10   11   12   13   14   15
```

Use both the negative lookbehind and negative lookahead operators together to precede only the single-digit numbers with zero:

```
regexprep(n, '(?<!\d)(\d)(?!\d)', '0$1')
ans =
   05    06    07    08    09   10   11   12   13   14   15
```

Using the Lookbehind Operator — (?<=test)expr

Example 1 — Positive and Negative Lookbehind Operators. Using the lookbehind operator, find the letter `r` that is preceded by the letter `u`:

```
str = 'Neural Network Toolbox';

startIndex = regexp(str, '(?<=u)r', 'start')
startIndex =
     4
```

Using the negative lookbehind operator, find the letter `r` that is *not* preceded by the letter `u`:

```
startIndex = regexp(str, '(?<!u)r', 'start')
startIndex =
    13
```

Example 2 — Lookbehind. Return the names and 7-digit telephone numbers for those people in the list that are in the 617 area code. The lookbehind `(?<=^617-)` finds those lines that begin with the number 617:

```
phone_list = {...
'978-389-2457 Kevin';       '617-922-3091 Ruth'; ...
'781-147-1748 Alan';        '508-643-9648 George'; ...
'617-774-6642 Lisa';        '617-241-0275 Greg'; ...
'413-995-9114 Jason';       '781-276-0482 Victoria'};
len = length(phone_list);

ph617 = regexp(phone_list, '(?<=^617-).*', 'match');

for k=1:len
str = char(ph617{k});
if ~isempty(str),   fprintf('   %s\n', str),   end
end
```

MATLAB returns the three numbers that have a 617 area code:

```
   922-3091 Ruth
   774-6642 Lisa
   241-0275 Greg
```

Using the Negative Lookbehind Operator— (?<!test)expr

Example — Negative Lookbehind. This example uses negative lookbehind to find those tasks that are not labelled as `Done` or `Pending`, Create a list of tasks, each with status information to the left:

```
tasks = {...
'ToDo     3892457';       'Done     9223091'; ...
'Pending  1471748';       'Maybe    7746642'; ...
'ToDo     2410275';       'Pending  4723596'; ...
'ToDo     9959114';       'Maybe    2760482'; ...
'ToDo     3080027';       'Done     1221941'};
count = length(tasks);
```

The regular expression looks for those task numbers that do not have a Done or Pending status. Note that you can use the or (|) operator in a lookaround to check for more than one condition:

```
doNow = regexp(tasks, '(?<!^(Done|Pending).*)\d+', 'match');
```

Now print out the results:

```
disp 'The following tasks need attention:'
for k=1:count
   s = char(doNow{k});
   if ~isempty(s),   fprintf('   %s\n', s),   end
end
```

The output displays all but the Done and Pending tasks:

```
The following tasks need attention:
   3892457
   7746642
   2410275
   9959114
   2760482
   3080027
```

Using Lookaround as a Logical Operator

One way in which a lookahead operation can be useful is to perform a logical AND between two conditions. This example initially attempts to locate all lowercase consonants in a text string. The text string is the first 50 characters of the M-file help for the normest function:

```
helptext = help('normest');
str = helptext(1:50)
str =
 NORMEST Estimate the matrix 2-norm.
    NORMEST(S
```

Merely searching for non-vowels ([^aeiouAEIOU]) does not return the expected answer, as the output includes capital letters, space characters, and punctuation:

```
c = regexp(str, '[^aeiouAEIOU]', 'match')
c =
  Columns 1 through 12
    ' '  'N'  'R'  'M'  'S'  'T'  ' '  's'  't'  'm' 't'

      -- etc. --
```

Try this again, using a lookahead operator to create the following AND condition:

```
(lowercase letter) AND (not a vowel).
```

This time, the result is correct:

```
c = regexp(str, '(?=[a-z])[^aeiou]', 'match')
c =
  's'  't'  'm '  't'  't'  'h'  'm'  't'  'r'  'x'
     'n'  'r'  'm'
```

Note that when using a lookahead operator to perform an AND, you need to place the match expression `expr` *after* the test expression `test`:

```
(?=test)expr or (?!test)expr
```

Quantifiers

With the quantifiers shown below, you can specify how many instances of an element are to be matched. The basic quantifying operators are listed in the first six rows of the table.

By default, MATLAB matches as much of an expression as possible. Using the operators shown in the last two rows of the table, you can override this default behavior. Specify these options by appending a `+` or `?` immediately following one of the six basic quantifying operators.

Operator	Usage
`expr{m,n}`	Must occur at least `m` times but no more than `n` times.
`expr{m,}`	Must occur at least `m` times.
`expr{n}`	Must match exactly `n` times. Equivalent to `{n,n}`.

Operator	Usage
expr?	Match the preceding element 0 times or 1 time. Equivalent to {0,1}.
expr*	Match the preceding element 0 or more times. Equivalent to {0,}.
expr+	Match the preceding element 1 or more times. Equivalent to {1,}.
q_expr+	Match as much of the quantified expression as possible, but do not rescan any portions of the string if the initial match fails. The term q_expr represents any of the expressions shown in the top six rows of this table.
q_expr?	Match only as much of the quantified expression as necessary. The term q_expr represents any of the expressions shown in the top six rows of this table. For an example, see "Lazy Quantifiers — expr*?" on page 2-78, below.

Zero or One — expr?

Use ? to make the HTML <code> and </code> tags optional in the string. The first string, hstr1, contains one occurrence of each tag. Since the expression uses ()? around the tags, one occurrence is a match:

```
hstr1 = '<td><a name="18854"></a><code>%%</code><br></td>';
expr = '</a>(<code>)?..(</code>)?<br>';

regexp(hstr1, expr, 'match')
ans =
    '</a><code>%%</code><br>'
```

The second string, hstr2, does not contain the code tags at all. Just the same, the expression matches because ()? allows for zero occurrences of the tags:

```
hstr2 = '<td><a name="18854"></a>%%<br></td>';
expr = '</a>(<code>)?..(</code>)?<br>';

regexp(hstr2, expr, 'match')
ans =
```

```
'</a>%%<br>'
```

Zero or More — expr*

The first `regexp` command looks for at least one occurrence of `<br>` and finds it. The second command parses a different string for at least one `<br>` and fails. The third command uses `*` to parse the same line for zero or more line breaks and this time succeeds.

```
hstr1 = '<p>This string has <br><br>line breaks</p>';
regexp(hstr1, '<p>.*(<br>).*</p>', 'match')
ans =
    '<p>This string has <br><br>line breaks</p>';

hstr2 = '<p>This string has no line breaks</p>';
regexp(hstr2, '<p>.*(<br>).*</p>', 'match')
ans =
     {}

regexp(hstr2, '<p>.*(<br>)*.*</p>', 'match')
ans =
    '<p>This string has no line breaks</p>';
```

One or More — expr+

Use `+` to verify that the HTML image source is not empty. This looks for one or more characters in the `gif` filename:

```
hstr = '<a href="s12.html"><img src="b_prev.gif" border=0>';
expr = '<img src="\w+.gif';

regexp(hstr, expr, 'match')
ans =
    '<img src="b_prev.gif'
```

Exact, Minimum, and Maximum Quantities — {min,max}

Use `{m}`, `{m,}`, and `{m,n}` to verify the `href` syntax used in HTML. This statement requires the `href` to have at least one non-whitespace character, followed by exactly one occurrence of `.html`, optionally followed by `#` and five to eight digits:

```
hstr = '<a name="18749"></a><a href="s13.html#18760">';
expr = '<a href="\w{1,}(\.html){1}(\#\d{5,8}){0,1}"';

regexp(hstr, expr, 'match')
ans =
    '<a href="s13.html#18760"'
```

Lazy Quantifiers — expr*?

This example shows the difference between the default (*greedy*) quantifier and the *lazy* quantifier (`?`). The first part of the example uses the default quantifier to match all characters from the opening `<tr` to the ending `</td`:

```
hstr = '<tr valign=top><td><a name="19184"></a><br></td>';
regexp(hstr, '</?t.*>', 'match')
ans =
    '<tr valign=top><td><a name="19184"></a><br></td>'
```

The second part uses the lazy quantifier to match the minimum number of characters between `<tr`, `<td`, or `</td` tags:

```
regexp(hstr, '</?t.*?>', 'match')
ans =
    '<tr valign=top>'    '<td>'    '</td>'
```

Tokens

Parentheses used in a regular expression not only group elements of that expression together, but also designate any matches found for that group as *tokens*. You can use tokens to match other parts of the same string. One advantage of using tokens is that they remember what they matched, so you can recall and reuse matched text in the process of searching or replacing.

This section covers

Operators Used with Tokens

Here are the operators you can use with tokens in MATLAB.

Operator	Usage
`(expr)`	Capture in a token all characters matched by the expression within the parentheses.
`\N`	Match the `N`th token generated by this command. That is, use `\1` to match the first token, `\2` to match the second, and so on.
`$N`	Insert the match for the `N`th token in the replacement string. Used only by the `regexprep` function. If `N` is equal to zero, then insert the entire match in the replacement string.
`(?(N)s1|s2)`	If `N`th token is found, then match `s1`, else match `s2`

Introduction to Using Tokens

You can turn any pattern being matched into a token by enclosing the pattern in parentheses within the expression. For example, to create a token for a dollar amount, you could use `'(\$\d+)'`. Each token in the expression is assigned a number, starting from 1, going from left to right. To make a reference to a token later in the expression, refer to it using a backslash followed by the token number. For example, when referencing a token generated by the third set of parentheses in the expression, use `\3`.

As a simple example, if you wanted to search for identical sequential letters in a string, you could capture the first letter as a token and then search for a matching character immediately afterwards. In the expression shown

below, the `(\S)` phrase creates a token whenever `regexp` matches any non-whitespace character in the string. The second part of the expression, `'\1'`, looks for a second instance of the same character immediately following the first:

```
poestr = ['While I nodded, nearly napping, ' ...
          'suddenly there came a tapping,'];

[mat tok ext] = regexp(poestr, '(\S)\1', 'match', ...
   'tokens', 'tokenExtents');
mat
mat =
    'dd'    'pp'    'dd'    'pp'
```

The tokens returned in cell array `tok` are:

```
'd', 'p', 'd', 'p'
```

Starting and ending indices for each token in the input string `poestr` are:

```
11 11,  26 26,  35 35,  57 57
```

Using Tokens — Example 1

Here is an example of how tokens are assigned values. Suppose that you are going to search the following text:

```
andy ted bob jim andrew andy ted mark
```

You choose to search the above text with the following search pattern:

```
and(y|rew)|(t)e(d)
```

This pattern has three parenthetical expressions that generate tokens. When you finally perform the search, the following tokens are generated for each match.

Match	Token 1	Token 2
andy	y	

Match	Token 1	Token 2
ted	t	d
andrew	rew	
andy	y	
ted	t	d

Only the highest level parentheses are used. For example, if the search pattern and(y|rew) finds the text andrew, token 1 is assigned the value rew. However, if the search pattern (and(y|rew)) is used, token 1 is assigned the value andrew.

Using Tokens — Example 2

Use (expr) and \N to capture pairs of matching HTML tags (e.g., <a> and <\a>) and the text between them. The expression used for this example is

```
expr = '<(\w+).*?>.*?</\1>';
```

The first part of the expression, ’<(\w+)’, matches an opening bracket (<) followed by one or more alphabetic, numeric, or underscore characters. The enclosing parentheses capture token characters following the opening bracket.

The second part of the expression, ’.*?>.*?’, matches the remainder of this HTML tag (characters up to the >), and any characters that may precede the next opening bracket.

The last part, '</\1>', matches all characters in the ending HTML tag. This tag is composed of the sequence </tag>, where tag is whatever characters were captured as a token.

```
hstr = '<!comment><a name="752507"></a><b>Default</b><br>';
expr = '<(\w+).*?>.*?</\1>';

[mat tok] = regexp(hstr, expr, 'match', 'tokens');
mat{:}
ans =
    <a name="752507"></a>
ans =
```

```
    <b>Default</b>

tok{:}
ans =
    'a'
ans =
    'b'
```

Tokens That Are Not Matched

For those tokens specified in the regular expression that have no match in the string being evaluated, `regexp` and `regexpi` return an empty string (`''`) as the token output, and an extent that marks the position in the string where the token was expected.

The example shown here executes `regexp` on the path string `str` returned from the MATLAB `tempdir` function. The regular expression `expr` includes six token specifiers, one for each piece of the path string. The third specifier `[a-z]+` has no match in the string because this part of the path, `Profiles`, begins with an uppercase letter:

```
str = tempdir
str =
   C:\WINNT\Profiles\bpascal\LOCALS~1\Temp\

expr = ['([A-Z]:)\\(WINNT)\\([a-z]+)?.*\\' ...
        '([a-z]+)\\([A-Z]+~\d)\\(Temp)\\'];

[tok ext] = regexp(str, expr, 'tokens', 'tokenExtents');
```

When a token is not found in a string, MATLAB still returns a token string and token extent. The returned token string is an empty character string (`''`). The first number of the extent is the string index that marks where the token was expected, and the second number of the extent is equal to one less than the first.

In the case of this example, the empty token is the third specified in the expression, so the third token string returned is empty:

```
tok{:}
```

```
ans =
    'C:'    'WINNT'     ''    'bpascal'    'LOCALS~1'    'Temp'
```

The third token extent returned in the variable ext has the starting index set to 10, which is where the nonmatching substring, Profiles, begins in the string. The ending extent index is set to one less than the starting index, or 9:

```
ext{:}
ans =
     1     2
     4     8
    10     9
    19    25
    27    34
    36    39
```

Using Tokens in a Replacement String

When using tokens in a replacement string, reference them using $1, $2, etc. instead of \1, \2, etc. This example captures two tokens and reverses their order. The first, $1, is 'Norma Jean' and the second, $2, is 'Baker'. Note that regexprep returns the modified string, not a vector of starting indices.

```
regexprep('Norma Jean Baker', '(\w+\s\w+)\s(\w+)', '$2, $1')
ans =
    Baker, Norma Jean
```

Named Capture

If you use a lot of tokens in your expressions, it may be helpful to assign them names rather than having to keep track of which token number is assigned to which token. Use the following operator to assign a name to a token that finds a match.

Operator	Usage
(?<name>expr)	Capture in a token all characters matched by the expression within the parentheses. Assign a name to the token.
\k<name>	Match the token referred to by name.

Operator	Usage
`$<name>`	Insert the match for named token in a replacement string. Used only with the `regexprep` function.
`(?(name)s1|s2)`	If named token is found, then match `s1`; otherwise, match `s2`

When referencing a named token within the expression, use the syntax `\k<name>` instead of the numeric `\1`, `\2`, etc.:

```
poestr = ['While I nodded, nearly napping, ' ...
          'suddenly there came a tapping,'];

regexp(poestr, '(?<anychar>.)\k<anychar>', 'match')
ans =
    'dd'    'pp'    'dd'    'pp'
```

Labeling Your Output

Named tokens can also be useful in labeling the output from the MATLAB regular expression functions. This is especially true when you are processing numerous strings.

This example parses different pieces of street addresses from several strings. A short name is assigned to each token in the expression string:

```
str1 = '134 Main Street, Boulder, CO, 14923';
str2 = '26 Walnut Road, Topeka, KA, 25384';
str3 = '847 Industrial Drive, Elizabeth, NJ, 73548';

p1 = '(?<adrs>\d+\s\S+\s(Road|Street|Avenue|Drive))';
p2 = '(?<city>[A-Z][a-z]+)';
p3 = '(?<state>[A-Z]{2})';
p4 = '(?<zip>\d{5})';

expr = [p1 ', ' p2 ', ' p3 ', ' p4];
```

As the following results demonstrate, you can make your output easier to work with by using named tokens:

```
loc1 = regexp(str1, expr, 'names')
loc1 =
     adrs: '134 Main Street'
     city: 'Boulder'
    state: 'CO'
      zip: '14923'

loc2 = regexp(str2, expr, 'names')
loc2 =
     adrs: '26 Walnut Road'
     city: 'Topeka'
    state: 'KA'
      zip: '25384'

loc3 = regexp(str3, expr, 'names')
loc3 =
     adrs: '847 Industrial Drive'
     city: 'Elizabeth'
    state: 'NJ'
      zip: '73548'
```

Conditional Expressions

With conditional expressions, you can tell MATLAB to match an expression only if a certain condition is true. A conditional expression is similar to an `if-then` or an `if-then-else` clause in programming. MATLAB first tests the state of a given condition, and the outcome of this tests determines what, if anything, is to be matched next. The following table shows the two conditional syntaxes you can use with MATLAB.

Operator	Usage
`(?(cond)expr)`	If condition `cond` is true, then match expression `expr`
`(?(cond)`$expr_1$`|`$expr_2$`)`	If condition `cond` is true, then match expression $expr_1$. Otherwise match expression $expr_2$

The first entry in this table is the same as an `if-then` statement. MATLAB tests the state of condition `cond` and then matches expression `expr` only if

the condition was found to be true. In the form of an `if-then` statement, it would look like this:

```
if cond then expr
```

The second entry in the table is the same as an `if-then-else` statement. If the condition is true, MATLAB matches $expr_1$; if false, it matches $expr_2$ instead. This syntax is equivalent to the following programming statement:

```
if cond then expr1 else expr2
```

The condition `cond` in either of these syntaxes can be any one of the following:

- A specific token, identified by either number or name, is located in the input string. See "Conditions Based on Tokens" on page 2-86, below.
- A lookaround operation results in a match. See "Conditions Based on a Lookaround Match" on page 2-87, below.
- A dynamic expression of the form `(?@cmd)` returns a nonzero numeric value. See "Conditions Based on Return Values" on page 2-88, below.

Conditions Based on Tokens

In a conditional expression, MATLAB matches the expression only if the condition associated with it is met. If the condition is based on a token, then the condition is met if MATLAB matches more than one character for the token in the input string.

To specify a token in a condition, use either the token number or, for tokens that you have assigned a name to, its name. Token numbers are determined by the order in which they appear in an expression. For example, if you specify three tokens in an expression (that is, if you enclose three parts of the expression in parentheses), then you would refer to these tokens in a condition statement as `1`, `2`, and `3`.

The following example uses the conditional statement `(?(1)her|his)` to match the string regardless of the gender used. You could translate this into the phrase, "**if** token `1` is found (i.e., `Mr` is followed by the letter `s`), **then** match `her`, **else** match `his`:

```
expr = 'Mr(s?)\..*?(?(1)her|his) son';
```

```
[mat tok] = regexp('Mr. Clark went to see his son', ...
   expr, 'match', 'tokens')
mat =
    'Mr. Clark went to see his son'
tok =
    {1x2 cell}

tok{:}
ans =
     ''    'his'
```

In the second part of the example, the token `s` is found and MATLAB matches the word `her`:

```
[mat tok] = regexp('Mrs. Clark went to see her son', ...
expr, 'match', 'tokens')
mat =
    'Mrs. Clark went to see her son'
tok =
    {1x2 cell}

tok{:}
ans =
    's'    'her'
```

Note When referring to a token within a condition, use just the number of the token. For example, refer to token `2` by using the number `2` alone, and not `\2` or `$2`.

Conditions Based on a Lookaround Match

Lookaround statements look for text that either precedes or follows an expression. If this lookaround text is located, then MATLAB proceeds to match the expression. You can also use lookarounds in conditional statements. In this case, if the lookaround text is located, then MATLAB considers the condition to be met and matches the associated expression. If the condition is not met, then MATLAB matches the `else` part of the expression.

Conditions Based on Return Values

MATLAB supports different types of dynamic expressions. One type of dynamic expression, having the form (?@cmd), enables you to execute a MATLAB command (shown here as cmd) while matching an expression. You can use this type of dynamic expression in a conditional statement if the command in the expression returns a numeric value. The condition is considered to be met if the return value is nonzero.

Dynamic Regular Expressions

In a dynamic expression, you can make the pattern that you want regexp to match dependent on the content of the input string. In this way, you can more closely match varying input patterns in the string being parsed. You can also use dynamic expressions in replacement strings for use with the regexprep function. This gives you the ability to adapt the replacement text to the parsed input.

You can include any number of dynamic expressions in the match_expr or replace_expr arguments of these commands:

```
regexp(string, match_expr)
regexpi(string, match_expr)
regexprep(string, match_expr, replace_expr)
```

MATLAB supports three types of dynamic operators for use in a match expression. See "Dynamic Operators for the Match Expression" on page 2-90 for more information.

Operator	Usage
(??expr)	Parse expr as a separate regular expression, and include the resulting string in the match expression. This gives you the same results as if you called regexprep inside of a regexp match expression.

Operator	Usage
`(?@cmd)`	Execute the MATLAB command `cmd`, discarding any output that may be returned. This is often used for diagnosing a regular expression.
`(??@cmd)`	Execute the MATLAB command `cmd`, and include the string returned by `cmd` in the match expression. This is a combination of the two dynamic syntaxes shown above: `(??expr)` and `(?@cmd)`.

MATLAB supports one type of dynamic expression for use in the replacement expression of a `regexprep` command. See "Dynamic Operators for the Replacement Expression" on page 2-95 for more information.

Operator	Usage
`${cmd}`	Execute the MATLAB command `cmd`, and include the string returned by `cmd` in the replacement expression.

Example of a Dynamic Expression

As an example of a dynamic expression, the following `regexprep` command correctly replaces the term `internationalization` with its abbreviated form, `i18n`. However, to use it on a different term such as `globalization`, you have to use a different replacement expression:

```
match_expr = '(^\w)(\w*)(\w$)';

replace_expr1 = '$118$3';
regexprep('internationalization', match_expr, replace_expr1)
ans =
    i18n

replace_expr2 = '$111$3';
regexprep('globalization', match_expr, replace_expr2)
ans =
    g11n
```

Using a dynamic expression `${num2str(length($2))}` enables you to base the replacement expression on the input string so that you do not have to

change the expression each time. This example uses the dynamic syntax `${cmd}` from the second table shown above:

```
match_expr = '(^\w)(\w*)(\w$)';
replace_expr = '$1${num2str(length($2))}$3';

regexprep('internationalization', match_expr, replace_expr)
ans =
    i18n

regexprep('globalization', match_expr, replace_expr)
ans =
    g11n
```

Dynamic Operators for the Match Expression

There are three types of dynamic expressions you can use when composing a match expression:

- "Dynamic Expressions that Modify the Match Expression — (??expr)" on page 2-91
- "Dynamic Commands that Modify the Match Expression — (??@cmd)" on page 2-91
- "Dynamic Commands that Serve a Functional Purpose — (?@cmd)" on page 2-92

The first two of these actually modify the match expression itself so that it can be made specific to changes in the contents of the input string. When MATLAB evaluates one of these dynamic statements, the results of that evaluation are included in the same location within the overall match expression.

The third operator listed here does not modify the overall expression, but instead enables you to run MATLAB commands during the parsing of a regular expression. This functionality can be useful in diagnosing your regular expressions.

Dynamic Expressions that Modify the Match Expression — (??expr). The `(??expr)` operator parses expression `expr`, and inserts the results back into the match expression. MATLAB then evaluates the modified match expression.

Here is an example of the type of expression that you can use with this operator:

```
str = {'5XXXXX', '8XXXXXXXX', '1X'};
regexp(str, '^(\d+)(??X{$1})$', 'match', 'once')
```

The purpose of this particular command is to locate a series of `X` characters in each of the strings stored in the input cell array. Note however that the number of `X`s varies in each string. If the count did not vary, you could use the expression `X{n}` to indicate that you want to match `n` of these characters. But, a constant value of `n` does not work in this case.

The solution used here is to capture the leading count number (e.g., the `5` in the first string of the cell array) in a token, and then to use that count in a dynamic expression. The dynamic expression in this example is `(??X{$1})`, where `$1` is the value captured by the token `\d+`. The operator `{$1}` makes a quantifier of that token value. Because the expression is dynamic, the same pattern works on all three of the input strings in the cell array. With the first input string, `regexp` looks for five `X` characters; with the second, it looks for eight, and with the third, it looks for just one:

```
regexp(str, '^(\d+)(??X{$1})$', 'match', 'once')
ans =
    '5XXXXX'    '8XXXXXXXX'    '1X'
```

Dynamic Commands that Modify the Match Expression — (??@cmd). MATLAB uses the `(??@function)` operator to include the results of a MATLAB command in the match expression. This command must return a string that can be used within the match expression.

The `regexp` command below uses the dynamic expression `(??@flilplr($1))` to locate a palindrome string, “Never Odd or Even”, that has been embedded into a larger string:

```
regexp(pstr, '(.{3,}).?(??@fliplr($1))', 'match')
```

The dynamic expression reverses the order of the letters that make up the string, and then attempts to match as much of the reversed-order string as possible. This requires a dynamic expression because the value for `$1` relies on the value of the token `(.{3,})`:

```
% Put the string in lowercase.
str = lower(...
  'Find the palindrome Never Odd or Even in this string');

% Remove all nonword characters.
str = regexprep(str, '\W*', '')
str =
   findthepalindromeneveroddoreveninthisstring

% Now locate the palindrome within the string.
palstr = regexp(str, '(.{3,}).?(??@fliplr($1))', 'match')
str =
   'neveroddoreven'
```

Dynamic expressions in MATLAB have access to the currently active workspace. This means that you can change any of the functions or variables used in a dynamic expression just by changing variables in the workspace. Repeat the last command of the example above, but this time define the function to be called within the expression using a function handle stored in the base workspace:

```
fun = @fliplr;

palstr = regexp(str, '(.{3,}).?(??@fun($1))', 'match')
palstr =
   'neveroddoreven'
```

Dynamic Commands that Serve a Functional Purpose — (?@cmd). The `(?@cmd)` operator specifies a MATLAB command that `regexp` or `regexprep` is to run while parsing the overall match expression. Unlike the other dynamic expressions in MATLAB, this operator does not alter the contents of the expression it is used in. Instead, you can use this functionality to get MATLAB to report just what steps it's taking as it parses the contents of one of your regular expressions.

The following example parses a word for zero or more characters followed by two identical characters followed again by zero or more characters:

```
regexp('mississippi', '\w*(\w)\1\w*', 'match')
ans =
    'mississippi'
```

To track the exact steps that MATLAB takes in determining the match, the example inserts a short script `(?@disp($1))` in the expression to display the characters that finally constitute the match. Because the example uses greedy quantifiers, MATLAB attempts to match as much of the string as possible. So, even though MATLAB finds a match toward the beginning of the string, it continues to look for more matches until it arrives at the very end of the string. From there, it backs up through the letters `i` then `p` and the next `p`, stopping at that point because the match is finally satisfied:

```
regexp('mississippi', '\w*(\w)(?@disp($1))\1\w*');
i
p
p
```

Now try the same example again, this time making the first quantifier lazy (`*?`). Again, MATLAB makes the same match:

```
regexp('mississippi', '\w*?(\w)\1\w*', 'match')
ans =
    'mississippi'
```

But by inserting a dynamic script, you can see that this time, MATLAB has matched the string quite differently. In this case, MATLAB uses the very first match it can find, and does not even consider the rest of the string:

```
regexp('mississippi', '\w*?(\w)(?@disp($1))\1\w*';)
m
i
s
```

To demonstrate how versatile this type of dynamic expression can be, consider the next example that progressively assembles a cell array as MATLAB iteratively parses the input string. The `(?!)` operator found at the end of the expression is actually an empty lookahead operator, and forces a failure at

each iteration. This forced failure is necessary if you want to trace the steps that MATLAB is taking to resolve the expression.

MATLAB makes a number of passes through the input string, each time trying another combination of letters to see if a fit better than last match can be found. On any passes in which no matches are found, the test results in an empty string. The dynamic script (?@if(~isempty($&))) serves to omit these strings from the matches cell array:

```
matches = {};
expr = ['(Euler\s)?(Cauchy\s)?(Boole)?(?@if(~isempty($&)),' ...
   'matches{end+1}=$&;end)(?!)'];

regexp('Euler Cauchy Boole', expr);

matches
matches =
    'Euler Cauchy Boole'    'Euler Cauchy '    'Euler '
'Cauchy Boole'    'Cauchy '    'Boole'
```

The operators $& (or the equivalent $0), $`, and $' refer to that part of the input string that is currently a match, all characters that precede the current match, and all characters to follow the current match, respectively. These operators are sometimes useful when working with dynamic expressions, particularly those that employ the (?@cmd) operator.

This example parses the input string looking for the letter g. At each iteration through the string, regexp compares the current character with g, and not finding it, advances to the next character. The example tracks the progress of scan through the string by marking the current location being parsed with a ^ character.

(The $` and $' operators capture that part of the string that precedes and follows the current parsing location. You need two single-quotation marks ($'') to express the sequence $' when it appears within a string.)

```
str = 'abcdefghij';
expr = '(?@disp(sprintf(''starting match: [%s^%s]'',$`,$'')))g';

regexp(str, expr, 'once');
```

```
starting match: [^abcdefghij]
starting match: [a^bcdefghij]
starting match: [ab^cdefghij]
starting match: [abc^defghij]
starting match: [abcd^efghij]
starting match: [abcde^fghij]
starting match: [abcdef^ghij]
```

Dynamic Operators for the Replacement Expression

The three types of dynamic expressions discussed above can be used only in the match expression (second input) argument of the regular expression functions. MATLAB provides one more type of dynamic expression; this one is for use in a replacement string (third input) argument of the `regexprep` function.

Dynamic Commands that Modify the Replacement Expression — ${cmd}. The `${cmd}` operator modifies the contents of a regular expression replacement string, making this string adaptable to parameters in the input string that might vary from one use to the next. As with the other dynamic expressions used in MATLAB, you can include any number of these expressions within the overall replacement expression.

In the `regexprep` call shown here, the replacement string is `'${convert($1,$2)}'`. In this case, the entire replacement string is a dynamic expression:

```
regexprep('This highway is 125 miles long', ...
          '(\d+\.?\d*)\W(\w+)', '${convert($1,$2)}')
```

The dynamic expression tells MATLAB to execute an M-file function named `convert` using the two tokens `(\d+\.?\d*)` and `(\w+)`, derived from the string being matched, as input arguments in the call to `convert`. The replacement string requires a dynamic expression because the values of `$1` and `$2` are generated at runtime.

The following example defines the M-file named `convert` that converts measurements from imperial units to metric. To convert values from the string being parsed, `regexprep` calls the `convert` function, passing in values for the quantity to be converted and name of the imperial unit:

```
function valout  = convert(valin, units)
switch(units)
    case 'inches'
        fun = @(in)in .* 2.54;    uout = 'centimeters';
    case 'miles'
        fun = @(mi)mi .* 1.6093;  uout = 'kilometers';
    case 'pounds'
        fun = @(lb)lb .* 0.4536;  uout = 'kilograms';
    case 'pints'
        fun = @(pt)pt .* 0.4731;  uout = 'litres';
    case 'ounces'
        fun = @(oz)oz .* 28.35;   uout = 'grams';
end
val = fun(str2num(valin));
valout = [num2str(val) ' ' uout];
```

```
regexprep('This highway is 125 miles long', ...
          '(\d+\.?\d*)\W(\w+)', '${convert($1,$2)}')
ans =
   This highway is 201.1625 kilometers long
```

```
regexprep('This pitcher holds 2.5 pints of water', ...
          '(\d+\.?\d*)\W(\w+)', '${convert($1,$2)}')
ans =
   This pitcher holds 1.1828 litres of water
```

```
regexprep('This stone weighs about 10 pounds', ...
          '(\d+\.?\d*)\W(\w+)', '${convert($1,$2)}')
ans =
   This stone weighs about 4.536 kilograms
```

As with the `(??@ )` operator discussed in an earlier section, the `${ }` operator has access to variables in the currently active workspace. The following `regexprep` command uses the array `A` defined in the base workspace:

```
A = magic(3)
A =
```

```
     8     1     6
     3     5     7
     4     9     2

regexprep('The columns of matrix _nam are _val', ...
          {'_nam', '_val'}, ...
          {'A', '${sprintf(''%d%d%d '', A)}'})
ans =
The columns of matrix A are 834 159 672
```

String Replacement

The regexprep function enables you to replace a string that is identified by a regular expression with another string. The following syntax replaces all occurrences of the regular expression expr in string str with the string repstr. The new string is returned in s. If no matches are found, return string s is the same as input string str.

```
s = regexprep('str', 'expr', 'repstr')
```

The replacement string can include any ordinary characters and also any of the operators shown in the following table:

Operator	Usage
Operators from Character Representation on page 2-103 table	The character represented by the operator sequence
$`	That part of the input string that precedes the current match
$& or $0	That part of the input string that is currently a match
$´	That part of the input string that follows the current match. In MATLAB, use $'' to represent the character sequence $´.
$N	The string represented by the token identified by name

Operator	Usage
$<name>	The string represented by the token identified by `name`
`${cmd}`	The string returned when MATLAB executes the command `cmd`

You can capture parts of the input string as tokens and then reuse them in the replacement string. Specify the parts of the string to capture using the token capture operator `(...)`. Specify the tokens to use in the replacement string using the operators `$1`, `$2`, `$N` to reference the first, second, and `N`th tokens captured. (See the section on "Tokens" on page 2-78 and the example "Using Tokens in a Replacement String" on page 2-83 in this documentation for information on using tokens.)

Note When referring to a token within a replacement string, use the number of the token preceded by a dollar sign. For example, refer to token `2` by using `$2`, and not `2` or `\2`.

The following example uses both the `${cmd}` and `$N` operators in the replacement strings of nested `regexprep` commands to capitalize the first letter of each sentence. The inner `regexprep` looks for the start of the entire string and capitalizes the single instance; the outer `regexprep` looks for the first letter following a period and capitalizes the two instances:

```
s1 = 'here are a few sentences.';
s2 = 'none are capitalized.';
s3 = 'let''s change that.';
str = [s1 ' ' s2 ' ' s3]

regexprep(regexprep(str, '(^.)', '${upper($1)}'), ...
   '(?<=\.\s*)([a-z])','${upper($1)}')

ans =
Here are a few sentences. None are capitalized. Let's change that.
```

Make `regexprep` more specific to your needs by specifying any of a number of options with the command. See the `regexprep` reference page for more information on these options.

Handling Multiple Strings

You can use any of the MATLAB regular expression functions with cell arrays of strings as well as with single strings. Any or all of the input parameters (the string, expression, or replacement string) can be a cell array of strings. The `regexp` function requires that the string and expression arrays have the same number of elements. The `regexprep` function requires that the expression and replacement arrays have the same number of elements. (The cell arrays do not have to have the same shape.)

Whenever either input argument in a call to `regexp`, or the first input argument in a call to `regexprep` function is a cell array, all output values are cell arrays of the same size.

This section covers the following topics:

Finding a Single Pattern in Multiple Strings

The example shown here uses the `regexp` function on a cell array of strings `cstr`. It searches each string of the cell array for consecutive matching letters (e.g., `'oo'`). The function returns a cell array of the same size as the input array. Each row of the return array contains the indices for which there was a match against the input cell array.

Here is the input cell array:

```
cstr = {                                  ...
'Whose woods these are I think I know.' ; ...
'His house is in the village though;'   ; ...
'He will not see me stopping here'      ; ...
'To watch his woods fill up with snow.'};
```

Find consecutive matching letters by capturing a letter as a token `(.)` and then repeating that letter as a token reference, `\1`:

```
idx = regexp(cstr, '(.)\1');

whos idx
  Name      Size                    Bytes  Class

  idx       4x1                       296  cell array

idx{:}
ans =                % 'Whose woods these are I think I know.'
    8                %          |8

ans =                % 'His house is in the village though;'
   23                %                          |23

ans =                % 'He will not see me stopping here'
    6    14    23    %        |6       |14      |23

ans =                % 'To watch his woods fill up with snow.'
   15    22          %                 |15    |22
```

To return substrings instead of indices, use the `'match'` parameter:

```
mat = regexp(cstr, '(.)\1', 'match');
mat{3}
ans =
   'll'    'ee'    'pp'
```

Finding Multiple Patterns in Multiple Strings

This example uses a cell array of strings in both the input string and the expression. The two cell arrays are of different shapes: `cstr` is 4-by-1 while `expr` is 1-by-4. The command is valid as long as they both have the same number of cells.

Find uppercase or lowercase `'i'` followed by a white-space character in `str{1}`, the sequence `'hou'` in `str{2}`, two consecutive matching letters in `str{3}`, and words beginning with `'w'` followed by a vowel in `str{4}`.

```
expr = {'i\s', 'hou', '(.)\1', '\<w[aeiou]'};
idx = regexpi(cstr, expr);

idx{:}
ans =                % 'Whose woods these are I think I know.'
   23    31          %                        |23     |31

ans =                % 'His house is in the village though;'
    5    30          %      |5                       |30

ans =                % 'He will not see me stopping here'
    6    14    23    %       |6      |14      |23

ans =                % 'To watch his woods fill up with snow.'
    4    14    28    %     |4        |14            |28
```

Note that the returned cell array has the dimensions of the input string, `cstr`. The dimensions of the return value are always derived from the input string, whenever the input string is a cell array. If the input string is not a cell array, then it is the dimensions of the expression that determine the shape of the return array.

Replacing Multiple Strings

When replacing multiple strings with `regexprep`, use a single replacement string if the expression consists of a single string. This example uses a common replacement value (`'--'`) for all matches found in the multiple string input `cstr`. The function returns a cell array of strings having the same dimensions as the input cell array:

```
s = regexprep(cstr, '(.)\1', '--', 'ignorecase')
s =
    'Whose w--ds these are I think I know.'
    'His house is in the vi--age though;'
    'He wi-- not s-- me sto--ing here'
    'To watch his w--ds fi-- up with snow.'
```

You can use multiple replacement strings if the expression consists of multiple strings. In this example, the input string and replacement string are both 4-by-1 cell arrays, and the expression is a 1-by-4 cell array. As long as the expression and replacement arrays contain the same number of

elements, the statement is valid. The dimensions of the return value match the dimensions of the input string:

```
expr = {'i\s', 'hou', '(.)\1', '\<w[aeiou]'};
repl = {'-1-'; '-2-'; '-3-'; '-4-'};

s = regexprep(cstr, expr, repl, 'ignorecase')
s =
    'Whose w-3-ds these are -1-think -1-know.'
    'His -2-se is in the vi-3-age t-2-gh;'
    'He -4--3- not s-3- me sto-3-ing here'
    'To -4-tch his w-3-ds fi-3- up -4-th snow.'
```

Operator Summary

MATLAB provides these operators for working with regular expressions:

- Character Types on page 2-102
- Character Representation on page 2-103
- "Grouping Operators" on page 2-66
- "Nonmatching Operators" on page 2-68
- "Positional Operators" on page 2-68
- Lookaround Operators on page 2-105
- Quantifiers on page 2-105
- Ordinal Token Operators on page 2-106
- Named Token Operators on page 2-106
- Conditional Expression Operators on page 2-107
- Dynamic Expression Operators on page 2-107
- Replacement String Operators on page 2-108

Character Types

Operator	Usage
.	Any single character, including white space

Character Types (Continued)

Operator	Usage
[$c_1c_2c_3$]	Any character contained within the brackets: c_1 or c_2 or c_3
[^$c_1c_2c_3$]	Any character not contained within the brackets: anything but c_1 or c_2 or c_3
[c_1-c_2]	Any character in the range of c_1 through c_2
\s	Any white-space character; equivalent to [\f\n\r\t\v]
\S	Any non-whitespace character; equivalent to [^ \f\n\r\t\v]
\w	Any alphabetic, numeric, or underscore character. For English character sets, this is equivalent to [a-zA-Z_0-9].
\W	Any character that is not alphabetic, numeric, or underscore. For English character sets, this is equivalent to [^a-zA-Z_0-9].
\d	Any numeric digit; equivalent to [0-9]
\D	Any nondigit character; equivalent to [^0-9]
\oN or \o{N}	Character of octal value N
\xN or \x{N}	Character of hexadecimal value N

Character Representation

Operator	Usage
\\	Backslash
\$	Dollar sign
\a	Alarm (beep)
\b	Backspace
\f	Form feed

Character Representation (Continued)

Operator	Usage
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\char`	If a character has special meaning in a regular expression, precede it with backslash (\) to match it literally.

Grouping Operators

Operator	Usage
`(expr)`	Group regular expressions and capture tokens.
`(?:expr)`	Group regular expressions, but do not capture tokens.
`(?>expr)`	Group atomically.
$\texttt{expr}_1 \vert \texttt{expr}_2$	Match expression $\texttt{expr}_1$ or expression $\texttt{expr}_2$.

Nonmatching Operators

Operator	Usage
`(?#comment)`	Insert a comment into the expression. Comments are ignored in matching.

Positional Operators

Operator	Usage
`^expr`	Match `expr` if it occurs at the beginning of the input string.
`expr$`	Match `expr` if it occurs at the end of the input string.

Positional Operators (Continued)

Operator	Usage
\<expr	Match expr when it occurs at the beginning of a word.
expr\>	Match expr when it occurs at the end of a word.
\<expr\>	Match expr when it represents the entire word.

Lookaround Operators

Operator	Usage
(?=expr)	Look ahead from current position and test if expr is found.
(?!expr)	Look ahead from current position and test if expr is not found
(?<=expr)	Look behind from current position and test if expr is found.
(?<!expr)	Look behind from current position and test if expr is not found.

Quantifiers

Operator	Usage
expr{m,n}	Match expr when it occurs at least m times but no more than n times consecutively.
expr{m,}	Match expr when it occurs at least m times consecutively.
expr{n}	Match expr when it occurs exactly n times consecutively. Equivalent to {n,n}.
expr?	Match expr when it occurs 0 times or 1 time. Equivalent to {0,1}.
expr*	Match expr when it occurs 0 or more times consecutively. Equivalent to {0,}.

Quantifiers (Continued)

Operator	Usage
expr+	Match expr when it occurs 1 or more times consecutively. Equivalent to {1,}.
q_expr*	Match as much of the quantified expression as possible, where q_expr represents any of the expressions shown in the first six rows of this table.
q_expr+	Match as much of the quantified expression as possible, but do not rescan any portions of the string if the initial match fails.
q_expr?	Match only as much of the quantified expression as necessary.

Ordinal Token Operators

Operator	Usage
(expr)	Capture in a token all characters matched by the expression within the parentheses.
\N	Match the Nth token generated by this command. That is, use \1 to match the first token, \2 to match the second, and so on.
$N	Insert the match for the Nth token in the replacement string. Used only by the regexprep function. If N is equal to zero, then insert the entire match in the replacement string.
(?(N)s1|s2)	If N^{th} token is found, then match s1, else match s2

Named Token Operators

Operator	Usage
(?<name>expr)	Capture in a token all characters matched by the expression within the parentheses. Assign a name to the token.

Named Token Operators (Continued)

Operator	Usage
\k<name>	Match the token referred to by name.
$<name>	Insert the match for named token in a replacement string. Used only with the regexprep function.
(?(name)s1|s2)	If named token is found, then match s1; otherwise, match s2

Conditional Expression Operators

Operator	Usage
(?(cond)expr)	If condition cond is true, then match expression expr
(?(cond)expr$_1$|expr$_2$)	If condition cond is true, then match expression expr$_1$. Otherwise match expression expr$_2$

Dynamic Expression Operators

Operator	Usage
(??expr)	Parse expr as a separate regular expression, and include the resulting string in the match expression. This gives you the same results as if you called regexprep inside of a regexp match expression.
(??@cmd)	Execute the MATLAB command cmd, discarding any output that may be returned. This is often used for diagnosing a regular expression.
(?@cmd)	Execute the MATLAB command cmd, and include the string returned by cmd in the match expression. This is a combination of the two dynamic syntaxes shown above: (??expr) and (?@cmd).
${cmd}	Execute the MATLAB command cmd, and include the string returned by cmd in the replacement expression.

Replacement String Operators

Operator	Usage
Operators from Character Representation on page 2-103 table	The character represented by the operator sequence
$`	That part of the input string that precedes the current match
$& or $0	That part of the input string that is currently a match
$·	That part of the input string that follows the current match. In MATLAB, use $'' to represent the character sequence $·.
$N	The string represented by the token identified by name
$<name>	The string represented by the token identified by name
${cmd}	The string returned when MATLAB executes the command cmd

Symbol Reference

In this section...

This section does not include symbols used in arithmetic, relational, and logical operations. For a description of these symbols, see the top of the list. "Functions — Alphabetical List" in the MATLAB Help browser.

Asterisk — *

An asterisk in a filename specification is used as a wildcard specifier, as described below.

Filename Wildcard

Wildcards are generally used in file operations that act on multiple files or directories. They usually appear in the string containing the file or directory specification. MATLAB matches all characters in the name exactly except for the wildcard character *, which can match any one or more characters.

To locate all files with names that start with 'january_' and have a mat file extension, use

```
dir('january_*.mat')
```

You can also use wildcards with the who and whos functions. To get information on all variables with names starting with 'image' and ending with 'Offset', use

```
whos image*Offset
```

At — @

The @ sign signifies either a function handle constructor or a directory that supports a MATLAB class.

Function Handle Constructor

The @ operator forms a handle to either the named function that follows the @ sign, or to the anonymous function that follows the @ sign.

Function Handles in General. Function handles are commonly used in passing functions as arguments to other functions. Construct a function handle by preceding the function name with an @ sign:

```
fhandle = @myfun
```

You can read more about function handles in "Function Handles" on page 1-126.

Handles to Anonymous Functions. Anonymous functions give you a quick means of creating simple functions without having to create M-files each time. You can construct an anonymous function and a handle to that function using the syntax

```
fhandle = @(arglist) body
```

where `body` defines the body of the function and `arglist` is the list of arguments you can pass to the function.

See "Anonymous Functions" on page 4-3 for more information.

Class Directory Designator

A MATLAB class directory contains source files that define the methods and properties of a class. All MATLAB class directory names must begin with an `@` sign:

```
\@myclass\get.m
```

See the documentation on MATLAB Classes for more information.

Colon — :

The colon operator generates a sequence of numbers that you can use in creating or indexing into arrays. See"Generating a Numeric Sequence" for more information on using the colon operator.

Numeric Sequence Range

Generate a sequential series of regularly spaced numbers from `first` to `last` using the syntax `first:last`. For an incremental sequence from 6 to 17, use

```
N = 6:17
```

Numeric Sequence Step

Generate a sequential series of numbers, each number separated by a `step` value, using the syntax `first:step:last`. For a sequence from 2 through 38, stepping by 4 between each entry, use

```
N = 2:4:38
```

Indexing Range Specifier

Index into multiple rows or columns of a matrix using the colon operator to specify a range of indices:

```
B = A(7, 1:5);          % Read columns 1-5 of row 7.
B = A(4:2:8, 1:5);      % Read columns 1-5 of rows 4, 6, and 8.
B = A(:, 1:5);          % Read columns 1-5 of all rows.
```

Conversion to Column Vector

Convert a matrix or array to a column vector using the colon operator as a single index:

```
A = rand(3,4);
B = A(:);
```

Preserving Array Shape on Assignment

Using the colon operator on the left side of an assignment statement, you can assign new values to array elements without changing the shape of the array:

```
A = rand(3,4);
A(:) = 1:12;
```

Comma — ,

A comma is used to separate the following types of elements.

Row Element Separator

When constructing an array, use a comma to separate elements that belong in the same row:

```
A = [5.92, 8.13, 3.53]
```

Array Index Separator

When indexing into an array, use a comma to separate the indices into each dimension:

```
X = A(2, 7, 4)
```

Function Input and Output Separator

When calling a function, use a comma to separate output and input arguments:

```
function [data, text] = xlsread(file, sheet, range, mode)
```

Command or Statement Separator

To enter more than one MATLAB command or statement on the same line, separate each command or statement with a comma:

```
for k = 1:10,   sum(A(k)),   end
```

Curly Braces — { }

Use curly braces to construct or get the contents of cell arrays.

Cell Array Constructor

To construct a cell array, enclose all elements of the array in curly braces:

```
C = {[2.6 4.7 3.9], rand(8)*6, 'C. Coolidge'}
```

Cell Array Indexing

Index to a specific cell array element by enclosing all indices in curly braces:

```
A = C{4,7,2}
```

See the documentation on Cell Arrays for more information.

Dot — .

The single dot operator has the following different uses in MATLAB.

Structure Field Definition

Add fields to a MATLAB structure by following the structure name with a dot and then a field name:

```
funds(5,2).bondtype = 'Corporate';
```

See the documentation on "Structures" on page 1-70 for more information.

Object Method Specifier

Specify the properties of an instance of a MATLAB class using the object name followed by a dot, and then the property name:

```
val = asset.current_value
```

See "Defining Your Own Classes" on page 1-165 for more information.

Dot-Dot — ..

Two dots in sequence refer to the parent of the current directory.

Parent Directory

Specify the directory immediately above your current directory using two dots. For example, to go up two levels in the directory tree and down into the `testdir` directory, use

```
cd ..\..\testdir
```

Dot-Dot-Dot (Ellipsis) — ...

A series of three consecutive periods (`...`) is the line continuation operator in MATLAB. This is often referred to as an *ellipsis*, but it should be noted that the line continuation operator is a three-character operator and is different from the single-character ellipsis represented in ASCII by the hexadecimal number 2026.

Line Continuation

Continue any MATLAB command or expression by placing an ellipsis at the end of the line to be continued:

```
sprintf('The current value of %s is %d', ...
        vname, value)
```

Entering Long Strings. You cannot use an ellipsis within single quotes to continue a string to the next line:

```
string = 'This is not allowed and will generate an ...
   error in MATLAB.'
```

To enter a string that extends beyond a single line, piece together shorter strings using either the concatenation operator (`[]`) or the `sprintf` function.

Here are two examples:

```
quote1 = [
 'Tiger, tiger, burning bright in the forests of the night,' ...
 'what immortal hand or eye could frame thy fearful symmetry?'];
quote2 = sprintf('%s%s%s', ...
 'In Xanadu did Kubla Khan a stately pleasure-dome decree,', ...
 'where Alph, the sacred river, ran ', ...
 'through caverns measureless to man down to a sunless sea.');
```

Dot-Parentheses — .( )

Use dot-parentheses to specify the name of a dynamic structure field.

Dynamic Structure Fields

Sometimes it is useful to reference structures with field names that can vary. For example, the referenced field might be passed as an argument to a function. Dynamic field names specify a variable name for a structure field.

The variable `fundtype` shown here is a dynamic field name:

```
type = funds(5,2).(fundtype);
```

See on page 80 for more information.

Exclamation Point — !

The exclamation point precedes operating system commands that you want to execute from within MATLAB.

Shell Escape

The exclamation point initiates a shell escape function. Such a function is to be performed directly by the operating system:

```
!rmdir oldtests
```

See "Shell Escape Functions" on page 2-7 for more information.

Parentheses — ( )

Parentheses are used mostly for indexing into elements of an array or for specifying arguments passed to a called function.

Array Indexing

When parentheses appear to the right of a variable name, they are indices into the array stored in that variable:

```
A(2, 7, 4)
```

Function Input Arguments

When parentheses follow a function name in a function declaration or call, the enclosed list contains input arguments used by the function:

```
function sendmail(to, subject, message, attachments)
```

Percent — %

The percent sign is most commonly used to indicate nonexecutable text within the body of a program. This text is normally used to include comments in your code. Some functions also interpret the percent sign as a conversion specifier.

See "Help Text" on page 3-11 for more information.

Single Line Comments

Precede any one-line comments in your code with a percent sign. MATLAB does not execute anything that follows a percent sign (that is, unless the sign is quoted, `'%'`):

```
% The purpose of this routine is to compute
% the value of ...
```

Conversion Specifiers

Some functions, like `sscanf` and `sprintf`, precede conversion specifiers with the percent sign:

```
sprintf('%s = %d', name, value)
```

Percent-Brace — %{ %}

The `%{` and `%}` symbols enclose a block of comments that extend beyond one line.

Block Comments

Enclose any multiline comments with percent followed by an opening or closing brace.

```
%{
The purpose of this routine is to compute
the value of ...
%}
```

Note With the exception of whitespace characters, the `%{` and `%}` operators must appear alone on the lines that immediately precede and follow the block of help text. Do not include any other text on these lines.

Semicolon — ;

The semicolon can be used to construct arrays, suppress output from a MATLAB command, or to separate commands entered on the same line.

Array Row Separator

When used within square brackets to create a new array or concatenate existing arrays, the semicolon creates a new row in the array:

```
A = [5, 8; 3, 4]
A =
      5      8
      3      4
```

Output Suppression

When placed at the end of a command, the semicolon tells MATLAB not to display any output from that command. In this example, MATLAB does not display the resulting 100-by-100 matrix:

```
A = ones(100, 100);
```

Command or Statement Separator

Like the comma operator, you can enter more than one MATLAB command on a line by separating each command with a semicolon. MATLAB suppresses output for those commands terminated with a semicolon, and displays the output for commands terminated with a comma.

In this example, assignments to variables `A` and `C` are terminated with a semicolon, and thus do not display. Because the assignment to `B` is comma-terminated, the output of this one command is displayed:

```
A = 12.5;  B = 42.7,  C = 1.25;
B =
   42.7000
```

Single Quotes — ' '

Single quotes are the constructor symbol for MATLAB character arrays.

Character and String Constructor

MATLAB constructs a character array from all characters enclosed in single quotes. If only one character is in quotes, then MATLAB constructs a 1-by-1 array:

```
S = 'Hello World'
```

See "Characters and Strings" on page 1-39 for more information.

Space Character

The space character serves a purpose similar to the comma in that it can be used to separate row elements in an array constructor, or the values returned by a function.

Row Element Separator

You have the option of using either commas or spaces to delimit the row elements of an array when constructing the array. To create a 1-by-3 array, use

```
A = [5.92 8.13 3.53]
A =
    5.9200    8.1300    3.5300
```

When indexing into an array, you must always use commas to reference each dimension of the array.

Function Output Separator

Spaces are allowed when specifying a list of values to be returned by a function. You can use spaces to separate return values in both function declarations and function calls:

```
function [data text] = xlsread(file, sheet, range, mode)
```

Slash and Backslash — / \

The slash (/) and backslash (\) characters separate the elements of a path or directory string. On Microsoft® Windows®-based systems, both slash and

backslash have the same effect. On The Open Group UNIX®-based systems, you must use slash only.

On a Windows system, you can use either backslash or slash:

```
dir([matlabroot '\toolbox\matlab\elmat\shiftdim.m'])
dir([matlabroot '/toolbox/matlab/elmat/shiftdim.m'])
```

On a UNIX system, use only the forward slash:

```
dir([matlabroot '/toolbox/matlab/elmat/shiftdim.m'])
```

Square Brackets — [ ]

Square brackets are used in array construction and concatenation, and also in declaring and capturing values returned by a function.

Array Constructor

To construct a matrix or array, enclose all elements of the array in square brackets:

```
A = [5.7, 9.8, 7.3; 9.2, 4.5, 6.4]
```

Concatenation

To combine two or more arrays into a new array through concatenation, enclose all array elements in square brackets:

```
A = [B, eye(6), diag([0:2:10])]
```

Function Declarations and Calls

When declaring or calling a function that returns more than one output, enclose each return value that you need in square brackets:

```
[data, text] = xlsread(file, sheet, range, mode)
```

Internal MATLAB Functions

In this section...

Overview

Many of the functions provided with the MATLAB software are implemented as M-files just like the M-files that you will create with MATLAB. Other MATLAB functions are precompiled executable programs called built-ins that run much more efficiently. Many of the MATLAB functions are also overloaded so that they handle different classes appropriately.

M-File Functions

If you look in the subdirectories of the `toolbox\matlab` directory, you can find the M-file sources to many of the functions supplied with MATLAB. You can locate your `toolbox\matlab` directory by typing

```
dir([matlabroot '\toolbox\matlab\'])
```

MATLAB functions with an M-file source are just like any other functions coded with MATLAB. When one of these M-file functions is called, MATLAB parses and executes each line of code in the M-file. It saves the parsed version of the function in memory, eliminating parsing time on any further calls to this function.

Identifying M-File Functions

To find out if a function is implemented with an M-file, use the `exist` function. The `exist` function searches for the name you enter on the MATLAB path and returns a number identifying the source. If the source is an M-file, then `exist` returns the number `2`. This example identifies the source for the `repmat` function as an M-file:

```
exist repmat
ans =
     2
```

The `exist` function also returns 2 for files that have a file type unknown to MATLAB. However, if you invoke `exist` on a MATLAB function name, the file type will be known to MATLAB and will return 2 only on M-files.

Viewing the Source Code

One advantage of functions implemented as M-files is that you can look at the source code. This may help when you need to understand why the function returns a value you did not expect, if you need to figure out how to code something in MATLAB that is already coded in a function, or perhaps to help you create a function that overloads one of the MATLAB functions.

To find the source code for any MATLAB M-file function, use `which`:

```
which repmat
   D:\matlabR14\toolbox\matlab\elmat\repmat.m
```

Built-In Functions

Functions that are frequently used or that can take more time to execute are often implemented as executable files. These functions are called *built-ins*.

Unlike M-file functions, you cannot see the source code for built-ins. Although most built-in functions do have an M-file associated with them, this file is there mainly to supply the help documentation for the function. Take the `reshape` function, for example, and find it on the MATLAB path:

```
which reshape
   D:\matlabR14\toolbox\matlab\elmat\reshape.m
```

If you type this M-file out, you will see that it consists almost entirely of help text. At the bottom is a call to the built-in executable image.

Identifying Built-In Functions

As with M-file functions, you can identify which functions are built-ins using the `exist` function. This function identifies built-ins by returning the number `5`:

```
exist reshape
ans =
     5
```

Forcing a Built-In Call

If you overload any of the MATLAB built-in functions to handle a specific class, then MATLAB will always call the overloaded function on that type. If, for some reason, you need to call the built-in version, you can override the usual calling mechanism using a function called `builtin`. The expression

```
builtin('reshape', arg1, arg2, ..., argN);
```

forces a call to MATLAB built-in `reshape`, passing the arguments shown, even though an overload exists for the class in this argument list.

Note With the exception of overloading, you should not create an M-file that has the same name as a MATLAB built-in. Because built-in functions are given a higher precedence than most other types of M-files (with the exception of private and subfunctions), MATLAB does not recognize M-file functions that share the same name with a built-in.

Overloaded MATLAB Functions

An overloaded function is an additional implementation of an existing function that has been designed specifically to handle a certain class. When you pass an argument of this type in a call to the function, MATLAB looks for the function implementation that handles that type and executes that function code.

Each overloaded MATLAB function has an M-file on the MATLAB path. The M-files for a certain class are placed in a directory named with an `@` sign followed by the class name. For example, to overload the MATLAB `plot`

function to plot expressions of a class named `polynom` differently than other class, you would create a directory called `@polynom` and store your own version of `plot.m` in that directory.

You can add your own overloads to any function by creating a class directory for the class you wish to support for that function, and creating an M-file that handles that type in a manner different from the default. See Defining Classes — Syntax and Developing Classes — Typical Workflow.

When you use the `which` command with the `-all` option, MATLAB returns all occurrences of the file you are looking for. This is an easy way to find functions that are overloaded:

```
which -all set         % Show all implementations for 'set'
```

Functions and Scripts

Program Development

In this section...

Overview

When you write a MATLAB function or script, you save it to a file called an M-file (named after its `.m` file extension). There are two types of M-files that you can write: scripts and functions. This section covers basic program development, describes how to write and call scripts and functions, and shows how to pass different types of data in a function call. Associated with each step of this process are certain MATLAB tools and utilities that are fully documented in the Desktop Tools and Development Environment documentation.

For more ideas on good programming style, see "Program Development" on page 12-18 in the MATLAB Programming Tips documentation. The Programming Tips section is a compilation of useful pieces of information that can show you alternate and often more efficient ways to accomplish common programming tasks while also expanding your knowledge of MATLAB.

Creating a Program

You can type in your program code using any text editor. This section focuses on using the MATLAB Editor/Debugger for this purpose. The Editor/Debugger is fully documented in Ways to Edit and Debug Files in the Desktop Tools and Development Environment documentation.

The first step in creating a program is to open an editing window. To create a new M-file, type the word `edit` at the MATLAB command prompt. To edit an existing M-file, type `edit` followed by the filename:

```
edit drawPlot.m
```

MATLAB opens a new window for entering your program code. As you type in your program, MATLAB keeps track of the line numbers in the left column.

Saving the Program

It is usually a good idea to save your program periodically while you are in the development process. To do this, click **File > Save** in the Editor/Debugger. Enter a filename with a `.m` extension in the **Save file as** dialog box that appears and click **OK**. It is customary and less confusing if you give the M-file the same name as the first function in the M-file.

Running the Program

Before trying to run your program, make sure that its M-file is on the MATLAB path. The MATLAB path defines those directories that you want MATLAB to know about when executing M-files. The path includes all the directories that contain functions provided with MATLAB. It should also include any directories that you use for your own functions.

Use the `which` function to see if your program is on the path:

```
which drawPlot
   D:\matlabR14\work\drawPlot.m
```

If not, add its directory to the path using the `addpath` function:

```
addpath('D:\matlabR14\work')
```

Now you can run the program just by typing the name of the M-file at the MATLAB command prompt:

```
drawPlot(xdata, ydata)
```

Getting the Bugs Out

In all but the simplest programs, you are likely to encounter some type of unexpected behavior when you run the program for the first time. Program defects can show up in the form of warning or error messages displayed in the command window, programs that hang (never terminate), inaccurate results, or some number of other symptoms. This is where the second functionality of the MATLAB Editor/Debugger becomes useful.

The MATLAB Debugger enables you to examine the inner workings of your program while you run it. You can stop the execution of your program at any point and then continue from that point, stepping through the code line by line and examining the results of each operation performed. You have the choice of operating the debugger from the Editor window that displays your program, from the MATLAB command line, or both.

The Debugging Process

You can step through the program right from the start if you want. For longer programs, you will probably save time by stopping the program somewhere in the middle and stepping through from there. You can do this by approximating where the program code breaks and setting a stopping point (or *breakpoint*) at that line. Once a breakpoint has been set, start your program from the MATLAB command prompt. MATLAB opens an Editor/Debugger window (if it is not already open) showing a green arrow pointing to the next line to execute.

From this point, you can examine any values passed into the program, or the results of each operation performed. You can step through the program line by line to see which path is taken and why. You can step into any functions that your program calls, or choose to step over them and just see the end results. You can also modify the values assigned to a variable and see how that affects the outcome.

To learn about using the MATLAB Debugger, see Debugging and Improving M-Files in the Desktop Tools and Development Environment documentation. Type `help debug` for a listing of all MATLAB debug functions.

For programming tips on how to debug, see "Debugging" on page 12-21.

Cleaning Up the Program

Even after your program is bug-free, there are still some steps you can take to improve its performance and readability. The MATLAB M-Lint utility generates a report that can highlight potential problems in your code. For example, you may be using the elementwise AND operator (`&`) where the short-circuit AND (`&&`) is more appropriate. You may be using the `find` function in a context where logical subscripting would be faster.

MATLAB offers M-Lint and several other reporting utilities to help you make the finishing touches to your program code. These tools are described under Tuning and Refining M-Files in the Desktop Tools and Development Environment documentation.

Improving Performance

The MATLAB Profiler generates a report that shows how your program spends its processing time. For details about using the MATLAB Profiler, see Profiling for Improving Performance in the MATLAB Desktop Tools and Development Environment documentation. For tips on other ways to improve the performance of your programs, see Chapter 10, “Performance”.

Three types of reports are available:

- “Summary Report” on page 3-5
- “Detail Report” on page 3-5
- “File Listing” on page 3-6

Summary Report

The summary report provides performance information on your main program and on every function it calls. This includes how many times each function is called, the total time spent in that function, along with a bar graph showing the relative time spent by each function.

Detail Report

When you click a function name in the summary report, MATLAB displays a detailed report on that function. This report shows the lines of that function that take up the most time, the time spent executing that line, the percentage

of total time for that function that is spent on that line, and a bar graph showing the relative time spent on the line.

File Listing

The detail report for a function also displays the entire M-file code for that function. This listing enables you to view the time-consuming code in the context of the entire function body. For every line of code that takes any significant time, additional performance information is provided by the statistics and by the color and degree of highlighting of the program code.

Checking It In

Source control systems offer a way to manage large numbers of files while they are under development. They keep track of the work done on these files as your project progresses, and also ensure that changes are made in a secure and orderly fashion.

If you have a source control system available to you, you will probably want to check your M-files into the system once they are complete. If further work is required on one of those files, you just check it back out, make the necessary modifications, and then check it back in again.

MATLAB provides an interface to external source control systems so that you can check files in and out directly from your MATLAB session. See Revision Control in the Desktop Tools and Development Environment documentation for instructions on how to use this interface.

Working with M-Files

In this section...

Overview

The MATLAB software provides a full programming language that enables you to write a series of MATLAB statements into a file and then execute them with a single command. You write your program in an ordinary text file, giving the file a name of `filename.m`. The term you use for `filename` becomes the new command that MATLAB associates with the program. The file extension of `.m` makes this a MATLAB M-file.

Types of M-Files

M-files can be *scripts* that simply execute a series of MATLAB statements, or they can be *functions* that also accept input arguments and produce output.

MATLAB scripts:

- Are useful for automating a series of steps you need to perform many times.
- Do not accept input arguments or return output arguments.
- Store variables in a workspace that is shared with other scripts and with the MATLAB command line interface.

MATLAB functions:

- Are useful for extending the MATLAB language for your application.

- Can accept input arguments and return output arguments.
- Store variables in a workspace internal to the function.

Basic Parts of an M-File

This simple function shows the basic parts of an M-file. Note that any line that begins with `%` is not executable:

```
function f = fact(n)                        Function definition line
% Compute a factorial value.                H1 line
% FACT(N) returns the factorial of N,       Help text
% usually denoted by N!

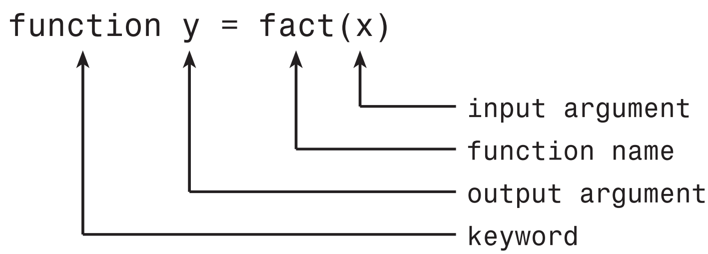
% Put simply, FACT(N) is PROD(1:N).         Comment
f = prod(1:n);                              Function body
```

The table below briefly describes each of these M-file parts. Both functions and scripts can have all of these parts, except for the function definition line which applies to functions only. These parts are described in greater detail following the table.

M-File Element	Description
Function definition line (functions only)	Defines the function name, and the number and order of input and output arguments
H1 line	A one line summary description of the program, displayed when you request `help` on an entire directory, or when you use `lookfor`
Help text	A more detailed description of the program, displayed together with the H1 line when you request `help` on a specific function
Function or script body	Program code that performs the actual computations and assigns values to any output arguments
Comments	Text in the body of the program that explains the internal workings of the program

Function Definition Line

The function definition line informs MATLAB that the M-file contains a function, and specifies the argument calling sequence of the function. This line contains the `function` keyword and must always be the first line of a function M-file, with the exception of lines that are nonexecutable comments. The function definition line for the `fact` function is



All MATLAB functions have a function definition line that follows this pattern.

Function Name. Function names must begin with a letter, may contain any alphanumeric characters or underscores, and must be no longer than the maximum allowed length (returned by the function `namelengthmax`). Because variables must obey similar rules, you can use the `isvarname` function to check whether a function name is valid:

```
isvarname myfun
```

Function names also cannot be the same as any MATLAB keyword. Use the `iskeyword` function with no inputs to display a list of all keywords.

Although function names can be of any length, MATLAB uses only the first `N` characters of the name (where `N` is the number returned by the function `namelengthmax`) and ignores the rest. Hence, it is important to make each function name unique in the first `N` characters:

```
N = namelengthmax
N =
    63
```

Note Some operating systems may restrict file names to shorter lengths.

The name of the text file that contains a MATLAB function consists of the function name with the extension `.m` appended. For example,

```
average.m
```

If the filename and the function definition line name are different, the internal (function) name is ignored. Thus, if `average.m` is the file that defines a function named `computeAverage`, you would invoke the function by typing

```
average
```

Note While the function name specified on the function definition line does not have to be the same as the filename, it is best to use the same name for both to avoid confusion.

Function Arguments. If the function has multiple output values, enclose the output argument list in square brackets. Input arguments, if present, are enclosed in parentheses following the function name. Use commas to separate multiple input or output arguments. Here is the declaration for a function named `sphere` that has three inputs and three outputs:

```
function [x, y, z] = sphere(theta, phi, rho)
```

If there is no output, leave the output blank

```
function printresults(x)
```

or use empty square brackets:

```
function [] = printresults(x)
```

The variables that you pass to the function do not need to have the same name as those in the function definition line.

The H1 Line

The H1 line, so named because it is the first help text line, is a comment line immediately following the function definition line. Because it consists of comment text, the H1 line begins with a percent sign, `%`. For the `average` function, the H1 line is

```
% AVERAGE Mean of vector elements.
```

This is the first line of text that appears when a user types `help` *`functionname`* at the MATLAB prompt. Further, the `lookfor` function searches on and displays only the H1 line. Because this line provides important summary information about the M-file, it is important to make it as descriptive as possible.

Help Text

You can create online help for your M-files by entering help text on one or more consecutive comment lines at the start of your M-file program. MATLAB considers the first group of consecutive lines immediately following the H1 line that begin with `%` to be the online help text for the function. The first line without `%` as the left-most character ends the help.

The help text for the `average` function is

```
% AVERAGE(X), where X is a vector, is the mean of vector
% elements. Nonvector input results in an error.
```

When you type `help` *`functionname`* at the command prompt, MATLAB displays the H1 line followed by the online help text for that function. The help system ignores any comment lines that appear after this help block.

Note Help text in an M-file can be viewed at the MATLAB command prompt only (using `help functionname`). You cannot display this text using the MATLAB Help browser. You can, however, use the Help browser to get help on MATLAB functions and also to read the documentation on any MathWorks™ products.

The Function or Script Body

The function body contains all the MATLAB code that performs computations and assigns values to output arguments. The statements in the function body can consist of function calls, programming constructs like flow control and interactive input/output, calculations, assignments, comments, and blank lines.

For example, the body of the `average` function contains a number of simple programming statements:

```
[m,n] = size(x);
if (~((m == 1) || (n == 1)) || ...
   (m == 1 && n == 1))  % Flow control

   error('Input must be a vector') % Error message display
end
y = sum(x)/length(x);            % Computation and assignment
```

Comments

As mentioned earlier, comment lines begin with a percent sign (`%`). Comment lines can appear anywhere in an M-file, and you can append comments to the end of a line of code. For example,

```
% Add up all the vector elements.
y = sum(x)           % Use the sum function.
```

In addition to comment lines, you can insert blank lines anywhere in an M-file. Blank lines are ignored. However, a blank line can indicate the end of the help text entry for an M-file.

Block Comments. To write comments that require more than one line, use the block comment operators, `%{` and `%}`:

```
%{
This next block of code checks the number of inputs
passed in, makes sure that each input is a valid data
type, and then branches to start processing the data.
%}
```

Note The %{ and %} operators must appear alone on the lines that immediately precede and follow the block of help text. Do not include any other text on these lines.

Creating a Simple M-File

You create M-files using a text editor. MATLAB provides a built-in editor, but you can use any text editor you like. Once you have written and saved the M-file, you can run the program as you would any other MATLAB function or command.

The process looks like this:

1 Create an M-file using a text editor.

```
function c = myfile(a,b)
c = sqrt((a.^2)+(b.^2))
```

2 Call the M-file from the command line, or from within another M-file.

```
a = 7.5
b = 3.342
c = myfile(a,b)

c =

    8.2109
```

Using Text Editors

M-files are ordinary text files that you create using a text editor. If you use the MATLAB Editor/Debugger, open a new file by selecting **New > M-File** from the **File** menu at the top of the MATLAB Command Window.

Another way to edit an M-file is from the MATLAB command line using the `edit` function. For example,

```
edit foo
```

opens the editor on the file `foo.m`. Omitting a filename opens the editor on an untitled file.

You can create the `fact` function shown in "Basic Parts of an M-File" on page 3-8 by opening your text editor, entering the lines shown, and saving the text in a file called `fact.m` in your current directory.

Once you have created this file, here are some things you can:

- List the names of the files in your current directory:

  ```
  what
  ```

- List the contents of M-file `fact.m`:

  ```
  type fact
  ```

- Call the `fact` function:

  ```
  fact(5)
  ans =
       120
  ```

A Word of Caution on Saving M-Files

Save any M-files you create and any MathWorks supplied M-files that you edit in directories outside of the directory tree in which the MATLAB software is installed. If you keep your files in any of the installed directories, your files may be overwritten when you install a new version of MATLAB.

MATLAB installs its software into directories under *`matlabroot`*`/toolbox`. To see what *`matlabroot`* is on your system, type `matlabroot` at the MATLAB command prompt.

Also note that locations of files in the *`matlabroot`*`/toolbox` directory tree are loaded and cached in memory at the beginning of each MATLAB session to

improve performance. If you save files to `matlabroot/toolbox` directories using an external editor, or if you add or remove files from these directories using file system operations, enter the commands `clear functionname` and `rehash toolbox` before you use the files in the current session.

For more information, see the `rehash` function reference page or the section Toolbox Path Caching in the Desktop Tools and Development Environment documentation.

Providing Help for Your Program

You can provide user information for the programs you write by including a help text section at the beginning of your M-file. (See “Help Text” on page 3-11).

You can also make help entries for an entire directory by creating a file with the special name `Contents.m` that resides in the directory. This file must contain only comment lines; that is, every line must begin with a percent sign. MATLAB displays the lines in a `Contents.m` file whenever you type

```
help directoryname
```

`Contents.m` files are optional. You might have directories of your own with M-files that you don’t necessarily want public. For this or other reasons, you might choose not to provide this type of help listing for these directories. If you have a directory that is on the path that does not have a `Contents.m` file, MATLAB displays `(No table of contents file)` for that directory in response to the `help` command. If you do not want to see this displayed, creating an empty `Contents.m` file will disable this message for that directory.

Also, if a directory does not contain a `Contents.m` file, typing

```
help directoryname
```

displays the first help line (the H1 line) for each M-file in the directory.

There is a tool in the Current Directory browser, called the Contents Report, that you can use to help create and validate your `Contents.m` files. See “Displaying and Updating a Report on the Contents of a Directory” in the

MATLAB Desktop Tools and Development Environment documentation for more information.

Cleaning Up the M-File When Done

When you have programmed all that you set out to do in your M-file, there is one last step to consider before it is complete. That is to make sure that you leave your program environment in a clean state that will not interfere with any other program code. For example, you might want to

- Close any files that you opened for import or export.
- Delete large temporary variables that take up significant space in memory.
- Lock or unlock memory to prevent or allow erasing M or MEX-files.
- Ensure that variables are not left in an unexpected state.
- Set your working directory back to its default if you have changed it.
- Make sure global and persistent variables are in the correct state.
- Restore any variables you used temporarily back to their original values.

MATLAB provides the `onCleanup` function for this purpose. This function, when used within any M-file program, establishes a cleanup routine for that function. When the function terminates, whether normally or due to an error, MATLAB automatically executes the cleanup routine. You can declare any number of cleanup routines for an M-file.

The following statement establishes a function handle to a cleanup routine, and associates the handle with output variable `cleanupObj`. (This variable is actually a MATLAB object.) If you clear `cleanupObj`, or when your function finishes executing, the function passed in as `@myCleanupRoutine` executes.

When your M-file program exits, MATLAB finds any instances of the `onCleanup` class and executes the associated function handles:

```
cleanupObj = onCleanup(@myCleanupRoutine);
```

Example 1

MATLAB closes the file with identifier `fid` when function `openFileSafely` terminates:

```
function openFileSafely(fileName)
fid = fopen(fileName, 'r');
c = onCleanup(@()fclose(fid));

s = fread(fid);
     .
     .
     .
% fclose(fid) executes here.
```

Example 2

This example preserves the current directory whether `functionThatMayError` returns an error or not:

```
function changeDirectorySafely(fileName)
   currentDir = pwd;
   c = onCleanup(@()cd(currentDir));

functionThatMayError;
   end   % c executes cd(currentDir) here
```

Creating P-Code Files

You can save a preparsed version of a function or script, called P-code files, for later MATLAB sessions using the `pcode` function. For example,

```
pcode average
```

parses `average.m` and saves the resulting pseudocode to the file named `average.p`. This saves MATLAB from reparsing `average.m` the first time you call it in each session.

MATLAB is very fast at parsing so the `pcode` function rarely makes much of a speed difference.

One situation where `pcode` does provide a speed benefit is for large GUI applications. In this case, many M-files must be parsed before the application becomes visible.

You can also use `pcode` to hide algorithms you have created in your M-file, if you need to do this for proprietary reasons.

M-File Scripts and Functions

In this section...

M-File Scripts

Scripts are the simplest kind of M-file because they have no input or output arguments. They are useful for automating series of MATLAB commands, such as computations that you have to perform repeatedly from the command line.

The Base Workspace

Scripts share the base workspace with your interactive MATLAB session and with other scripts. They operate on existing data in the workspace, or they can create new data on which to operate. Any variables that scripts create remain in the workspace after the script finishes so you can use them for further computations. You should be aware, though, that running a script can unintentionally overwrite data stored in the base workspace by commands entered at the MATLAB command prompt.

Simple Script Example

These statements calculate `rho` for several trigonometric functions of `theta`, then create a series of polar plots:

```
% An M-file script to produce       % Comment lines
% "flower petal" plots
theta = -pi:0.01:pi;                % Computations
rho(1,:) = 2 * sin(5 * theta) .^ 2;
rho(2,:) = cos(10 * theta) .^ 3;
rho(3,:) = sin(theta) .^ 2;
rho(4,:) = 5 * cos(3.5 * theta) .^ 3;
```

```
for k = 1:4
    polar(theta, rho(k,:))          % Graphics output
    pause
end
```

Try entering these commands in an M-file called `petals.m`. This file is now a MATLAB script. Typing `petals` at the MATLAB command line executes the statements in the script.

After the script displays a plot, press **Enter** or **Return** to move to the next plot. There are no input or output arguments; `petals` creates the variables it needs in the MATLAB workspace. When execution completes, the variables (`i`, `theta`, and `rho`) remain in the workspace. To see a listing of them, enter `whos` at the command prompt.

M-File Functions

Functions are program routines, usually implemented in M-files, that accept input arguments and return output arguments. You define MATLAB functions within a function M-file; that is, a file that begins with a line containing the `function` key word. You cannot define a function within a script M-file or at the MATLAB command line.

Functions always begin with a function definition line and end either with the first matching `end` statement, the occurrence of another function definition line, or the end of the M-file, whichever comes first. Using `end` to mark the end of a function definition is required only when the function being defined contains one or more nested functions.

Functions operate on variables within their own workspace. This workspace is separate from the base workspace; the workspace that you access at the MATLAB command prompt and in scripts.

The Function Workspace

Each M-file function has an area of memory, separate from the MATLAB base workspace, in which it operates. This area, called the function workspace, gives each function its own workspace context.

While using MATLAB, the only variables you can access are those in the calling context, be it the base workspace or that of another function. The variables that you pass to a function must be in the calling context, and the function returns its output arguments to the calling workspace context. You can, however, define variables as global variables explicitly, allowing more than one workspace context to access them. You can also evaluate any MATLAB statement using variables from either the base workspace or the workspace of the calling function using the `evalin` function. See "Extending Variable Scope" on page 2-17 for more information.

Simple Function Example

The `average` function is a simple M-file that calculates the average of the elements in a vector:

```
function y = average(x)
% AVERAGE Mean of vector elements.
% AVERAGE(X), where X is a vector, is the mean of vector
% elements. Nonvector input results in an error.
[m,n] = size(x);
if (~((m == 1) | (n == 1)) | (m == 1 & n == 1))
    error('Input must be a vector')
end
y = sum(x)/length(x);      % Actual computation
```

Try entering these commands in an M-file called `average.m`. The `average` function accepts a single input argument and returns a single output argument. To call the `average` function, enter

```
z = 1:99;

average(z)
ans =
    50
```

Types of Functions

MATLAB provides the following types of functions. Each function type is described in more detail in a later section of this documentation:

- The “Primary M-File Functions” on page 4-15 is the first function in an M-file and typically contains the main program.
- “Subfunctions” on page 4-33 act as subroutines to the main function. You can also use them to define multiple functions within a single M-file.
- “Nested Functions” on page 4-16 are functions defined within another function. They can help to improve the readability of your program and also give you more flexible access to variables in the M-file.
- “Anonymous Functions” on page 4-3 provide a quick way of making a function from any MATLAB expression. You can compose anonymous functions either from within another function or at the MATLAB command prompt.
- “Overloaded Functions” on page 4-37 are useful when you need to create a function that responds to different types of inputs accordingly. They are similar to overloaded functions in any object-oriented language.
- “Private Functions” on page 4-35 give you a way to restrict access to a function. You can call them only from an M-file function in the parent directory.

You might also see the term function functions in the documentation. This is not really a separate function type. The term function functions refers to any functions that accept another function as an input argument. You can pass a function to another function using a function handle.

Organizing Your Functions

When writing and saving your M-file functions, you have several options on how to organize the functions within the M-file, and also where in your directory structure you want to save them. Be sure to place your function M-files either in the directory in which you plan to run MATLAB, or in some other directory that is on the MATLAB path.

Use this table as a general guide when creating and saving your M-files:

If your program or routine . . .	then . . .
Requires only one function	Make it a single (primary) function in the M-file.

If your program or routine . . .	then . . .
Also requires subroutines	Make each subroutine a subfunction within same M-file as the primary.
Is for use only in the context of a certain function	Nest it within the other function. Nested functions also offer wider access to variables within the function.
Is a constructor or method of a MATLAB class	Put the M-file in a MATLAB class directory.
Is to have limited access	Put the M-file in a private subdirectory.
Is part of a group of similar functions or classes	Put the M-file in a package subdirectory.

If necessary, you can work around some of the constraints regarding function access by using function handles. You might find this useful when debugging your functions.

Identifying Dependencies

Most any program you write will make calls to other functions and scripts. If you need to know what other functions and scripts your program is dependent upon, use one of the techniques described below.

Simple Display of M-File Dependencies

For a simple display of all M-files referenced by a particular function, follow these steps:

1. Type `clear functions` to clear all functions from memory (see Note below).

> **Note** `clear functions` does not clear functions locked by `mlock`. If you have locked functions (which you can check using `inmem`) unlock them with `munlock`, and then repeat step 1.

2 Execute the function you want to check. Note that the function arguments you choose to use in this step are important, because you can get different results when calling the same function with different arguments.

3 Type `inmem` to display all M-files that were used when the function ran. If you want to see what MEX-files were used as well, specify an additional output:

```
[mfiles, mexfiles] = inmem
```

Detailed Display of M-File Dependencies

For a much more detailed display of dependent function information, use the `depfun` function. In addition to M-files, `depfun` shows which built-ins and classes a particular function depends on:

```
[list, builtins, classes] = depfun('strtok.m');

list
list =
    'D:\matlabR14\toolbox\matlab\strfun\strtok.m'
    'D:\matlabR14\toolbox\distcomp\toChar.m'
    'D:\matlabR14\toolbox\matlab\datafun\prod.m'
    'D:\matlabR14\toolbox\matlab\datatypes\@opaque\char.m'
              .
              .
              .
```

Calling Functions

In this section...

Command vs. Function Syntax

Overview

MATLAB syntax differs for issuing commands and for calling functions. This is sometimes referred to as *command-function duality*. In many cases, you can use either of the two syntaxes in commands and in function calls.

You are issuing a command when you tell MATLAB to do something. You might use a function name in the command and possibly a few switches or modifiers, usually expressed in the form of character strings. But, in most cases, there is no need to pass data such as numeric values or variables in a command. There is also no expectation that the command returns any output value.

A few examples of MATLAB commands are shown here. Note that MATLAB passes only string arguments and does not return any output:

```
clear
format long
dbstop in find_maxval
addpath C:\srcdir D:\matlab\testdir2 -end
whos -file savefile.mat -regexp ^p*
```

Unlike commands, most functions act upon data. You pass data into the function when calling it, and often expect it to return data upon completion. These input and output values can be in the form of any MATLAB class (e.g., arrays of integers, characters, cells, function handles, etc.).

Examples of function calls are shown below. Note that MATLAB requires parentheses around the input argument list, quotation marks around characters and strings, and brackets around multiple output values:

```
y = eye(n)
C = complex(A, B);
m = memmapfile('records.dat', 'Offset', 1024, ...
    'Format', 'uint32');
[x1 x2 x3 x4] = deal(A{:});
```

Regardless of which syntax you use, function names are always sensitive to case. When you call a function, use the correct combination of upper and lowercase letters so that the name is an exact match. Otherwise, you risk calling a different function that does match, but is elsewhere on the path.

MATLAB Command Syntax

A command that makes a function call consists of the function name followed by one or more arguments separated by spaces. In most cases, all input arguments are considered to be strings. Because of this, enclosing string arguments with quotation marks is optional.

The format for calling a function using command syntax is

```
functionname string1 string2 string3
```

You can use command syntax in calling a function when both of the following are true:

- All input arguments must be characters or strings. Variable names, expressions that require evaluation, and non-character classes (e.g., `doubles`, `structures`, `function handles`) are not allowed.
- You do not need to capture output in a variable. Commands that display information on your monitor screen are acceptable.

Note There are a few functions, such as `save` and `load`, that might appear to depart from this first rule. For an explanation of how these functions operate when using this syntax, see "Using save and load with Command Syntax" on page 3-28.

When you use command syntax, MATLAB interprets each input argument as a character string literal. There is no need to enclose these string arguments in quotation marks unless the argument includes one or more space characters. This is true whether the argument is a string of plain text, a file name, or a command switch:

```
strcat one two three four          % Command with 4 arguments.
ans =
   onetwothreefour

strcat 'one' 'two' 'three' 'four'  % Command with 4 arguments.
ans =
   onetwothreefour

strcat 'one two three four'        % Command with 1 argument.
ans =
   one two three four
```

Several examples of command syntax are given below.

Example 1. This command copies file `square.m` to directory `D:\matlab\functions`. All arguments are strings:

```
copyfile square.m D:\matlab\functions
```

Example 2. The example on the left calls `disp` using command syntax. MATLAB interprets `A` as a string literal and displays the character `A`.

The example on the right passes the *value* of `A` to `disp`, which then displays `3.1416`:

Command Syntax	Function Syntax

`A = pi;`
 	
`disp A`
`   A`
	`A = pi;`
	
	`disp(A)`
	`   3.1416`

Using save and load with Command Syntax. There are a few functions, such as `save` and `load`, that do accept variable names as input arguments. Examples follow:

```
save mydata.mat x y z        % x, y, and z are variables
load mydata x z              % x and z are variables
clear N                      % N is a variable
whos A                       % A is a variable
```

MATLAB Function Syntax

Function calls written in function syntax

- Enclose the input argument list in parentheses
- Separate the inputs with commas
- Enclose string arguments with single quotation marks
- Optionally assign any output from the function to one or more output arguments

Unlike command syntax, there are no limitations on when you can use function syntax in a function call.

Function calls written in function syntax enclose the input argument list in parentheses, separate the inputs with commas, enclose string arguments with single quotation marks, and optionally assign any output from the function to one or more output arguments. Unlike command syntax, there are no limitations on when you can use function syntax in a function call.

The format for MATLAB function syntax is

```
out = functionname(variable, 'string', expression, ...);
```

Calls written in function syntax pass the *values* assigned to each variable in the argument list. For example, this expression passes the values assigned to A0, A1, and A2 to the polyeig function:

```
e = polyeig(A0, A1, A2)
```

If a function returns more than one value, separate the output variables with commas or spaces, and enclose them all in square brackets ([ ]):

```
[out1, out2, ..., outN] = functionname(in1, in2, ..., inN);
```

For example,

```
[pathstr, name, ext, versn] = fileparts(filename);
```

Several examples of function syntax appear below. For more examples, see the section on "Common Mistakes In Syntax" on page 3-30

Example 1 — Simple Variable Comparison. Passing two variables representing equal strings to the strcmp function using function and command syntaxes gives different results. The function syntax passes the values of the arguments. strcmp returns a 1, which means they are equal:

```
str1 = 'one';       str2 = 'one';

strcmp(str1, str2)                  % Function syntax
ans =
   1       (equal)
```

The command syntax passes the names of the variables, 'str1' and 'str2', which are unequal:

```
str1 = 'one';       str2 = 'one';

strcmp str1 str2                    % Command syntax
ans =
   0       (unequal)
```

Example 2 — Passing Variable Names. The `reshape` function takes three input arguments: a variable name and two integers to specify dimensions for the new shape. It also returns the reshaped array. For both of these reasons, you need to use function syntax for this operation:

```
S1 = ...
  'MATLAB: Accelerating the pace of engineering and science.';

S2 = reshape(S1, 19, 3);
S2'
ans =
MATLAB: Acceleratin
g the pace of engin
eering and science.
```

Command syntax interprets all three input arguments as strings and provides no means for capturing the output:

```
reshape S1 19 3;
??? Error using ==> reshape
Size arguments must be integer scalars.
```

Common Mistakes In Syntax

The two MATLAB syntax styles are generally easy to use. You should have no difficulty in using them if you keep in mind the rules stated in the previous sections. Just the same, there are certain potential errors to watch out for.

In all examples in this section, it is the group of statements on the left that are incorrect, and the statements on the right that show the correct usage.

Example 1 — Numeric Values Evaluated As Strings. The statement on the left, below, appears to report that 500 is not numeric. However, because this statement uses command syntax, the input is actually the string '500' and not the number. Use function syntax, as shown on the right, to get the correct answer:

```
isnumeric 500                 isnumeric(500)
ans =                         ans =
    0                             1
```

Example 2 — Equal Values that Appear As Unequal. In this example, it might seem that MATLAB is reporting the values of variables `A` and `B` as unequal. However, it is not the *values* of `A` and `B` that are being compared here; it is the variable names `'A'` and `'B'`:

```
A = 500;   B = 500;              A = 500;   B = 500;
isequal A B                      isequal(A, B)
ans =                            ans =
     0                                1
```

Example 3 — Command Switches Used in Function Syntax. When using a command switch or modifier with function syntax, remember to enclose not only the input arguments in quotation marks, but the command switch, as well. In this example, `-file` needs to have quotation marks around it:

```
whos(-file, 'savefile.mat')      whos('-file', 'tempfile.mat')
```

A simpler method is to use command syntax for this type of statement:

```
whos -file tempfile.mat
```

Example 4 — Translation of Keywords. In command syntax, MATLAB interprets keywords in the same way it does variable names, as string literals. The statement to the left instructs MATLAB to search for a directory with the literal name `'matlabroot'`, when what was intended was the directory specified by this keyword. Function syntax uses the value of the keyword instead:

```
cd matlabroot                    cd(matlabroot)
```

Example 5 — Variables That Hold Filenames. This example, uses the `fopen` function to open the file `accounts.txt`. When this is done using command syntax, MATLAB looks for a file named `filename`, which does not exist. When working with filenames that are stored in variables, it is usually best to use function syntax:

```
filename = 'accounts.txt';       filename = 'accounts.txt';
fopen filename;                  fopen(filename);
```

Example 6 — Invalid String Comparisons. This example attempts to see if the class of vector `A` is an 8-bit unsigned integer (`uint8`), but the comparison is really between the strings `'class(A)'` and `'int8'`:

```
A = int8(1:8)                        A = int8(1:8)
strcmp class(A) int8                 strcmp(class(A), 'int8')
ans =                                ans =
     0                                    1
```

Example 7 — Numeric Arguments. This example shows that command syntax does not accept numeric arguments. Because command syntax assumes that each input argument is a character string, the numeric input 3.499 is interpreted by MATLAB as a five-element character array `'3.499'`, numerically equivalent to the vector [51 46 52 57 57].

```
round 3.499                          round(3.499)
ans =                                ans =
   51   46   52   57   57               3
```

Example 8 — Save and Load. The `save` and `load` functions are often easier to use with command syntax. The statement `save M` saves variable `M`, not the character `M`, to the workspace:

```
M = magic(20);                       M = magic(20);
save(M)                              save M          % or save('M')
clear M                              clear M
load(M)                              load M          % or load('M')
```

Example 9 — Class as a Command. When using the `class` function to display or return the class of a variable or value, always use the function syntax:

```
class pi                             class(pi)
ans =                                ans =
   char                                 double
```

Recognizing Function Calls That Use Command Syntax

It can be difficult to tell whether a MATLAB expression is a function call using command syntax or another kind of expression, such as an operation on one or more variables. Consider the following example:

```
ls ./d
```

Is this a call to the `ls` function with the directory `./d` as its argument? Or is it a request to perform elementwise division on the array that is the value of the `ls` variable, using the value of the `d` variable as the divisor?

This example might appear unambiguous because MATLAB can determine whether `ls` and `d` are functions or variables, but that is not always true. Some MATLAB components, such as M-Lint and the Editor/Debugger, must operate without reference to the MATLAB path or workspace. MATLAB therefore uses syntactic rules to determine when an expression is a function call using command syntax.

The rules are complicated and have exceptions. In general, when MATLAB recognizes an identifier (which might name a function or a variable), it analyzes the characters that follow the identifier to determine what kind of expression exists. The expression is usually a function call using command syntax when all of the following are true:

1. The identifier is followed immediately by white space.

2. The characters following the white space are not parentheses or an assignment operator.

3. The characters following the white space are not an operator that is itself followed by additional white space and then by characters that can legitimately follow an operator.

The example above meets all three criteria and is therefore a function call using command syntax:

```
ls ./d
```

The following examples are not function calls using command syntax:

```
% No white space following the ls identifier
% Interpretation: elementwise division
ls./d

% Parenthesis following white space
% Interpretation: function call using function syntax
```

```
ls ('./d')

% Assignment operator following white space
% Interpretation: assignment to a variable
ls =d

% Operator following white space, followed in turn by
% more white space and a variable
% Interpretation: elementwise division
ls ./ d
```

What Happens When You Call a Function

When you call a function M-file from either the command line or from within another M-file, MATLAB parses the function into pseudocode and stores it in memory. This prevents MATLAB from having to reparse a function each time you call it during a session. The pseudocode remains in memory until you clear it using the `clear` function, or until you quit MATLAB.

Clearing Functions from Memory

You can use `clear` in any of the following ways to remove functions from the MATLAB workspace.

Syntax	Description
`clear functionname`	Remove specified function from workspace.
`clear functions`	Remove all compiled M-functions.
`clear all`	Remove all variables and functions.

Determining Which Function Gets Called

When more than one function has the same name, which one does MATLAB call? This section explains the process that MATLAB uses to make this decision. It covers the following topics:

- "Function Scope" on page 3-35
- "Function Precedence Order" on page 3-35
- "Multiple Implementation Types" on page 3-37

- "Querying Which Function Gets Called" on page 3-37

Keep in mind that there are certain situations in which function names can conflict with *variables* of the same name. See "Potential Conflict with Function Names" on page 2-14 for more information.

Function Scope

Any functions you call must first be within the scope of (i.e., visible to) the calling function or your MATLAB session. MATLAB determines if a function is in scope by searching for the function's executable file according to a certain order (see Function Precedence Order, below).

One key part of this search order is the MATLAB path. The path is an ordered list of directories that MATLAB defines on startup. You can add or remove any directories you want from the path. MATLAB searches the path for the given function name, starting at the first directory in the path string and continuing until either the function file is found or the list of directories is exhausted. If no function of that name is found, then the function is considered to be out of scope and MATLAB issues an error.

Function Precedence Order

The function precedence order determines the precedence of one function over another based on the type of function and its location on the MATLAB path. MATLAB selects the correct function for a given context by applying the following function precedence rules in the order given here.

For items 3 through 7 in this list, the file MATLAB searches for can be any of four types: an M- or built-in file, preparsed M-file (P-Code), compiled C or Fortran file (MEX-file), or Simulink® model (MDL-file). See "Multiple Implementation Types" on page 3-37 for more on this.

1. Variable

   Before assuming that a name should match a function, MATLAB checks the current workspace to see if it matches a variable name. If MATLAB finds a match, it stops the search.

2. Subfunction

Subfunctions take precedence over all other M-file functions and overloaded methods that are on the path and have the same name. Even if the function is called with an argument of type matching that of an overloaded method, MATLAB uses the subfunction and ignores the overloaded method.

3 Private function

Private functions are called if there is no subfunction of the same name within the current scope. As with subfunctions, even if the function is called with an argument of type matching that of an overloaded method, MATLAB uses the private function and ignores the overloaded method.

4 Class constructor

Constructor functions (functions having names that are the same as the @ directory, for example `@polynom/polynom.m`) take precedence over other MATLAB functions. Therefore, if you create an M-file called `polynom.m` and put it on your path before the constructor `@polynom/polynom.m` version, MATLAB will always call the constructor version.

5 Overloaded method

MATLAB calls an overloaded method if it is not superseded by a subfunction or private function. Which overloaded method gets called depends on the classes of the objects passed in the argument list.

6 Function in the current directory

A function in the current working directory is selected before one elsewhere on the path.

7 Function elsewhere on the path

Finally, a function elsewhere on the path is selected. A function in a directory that is toward the beginning of the path string is given higher precedence.

Note Because variables have the highest precedence, if you have created a variable of the same name as a function, MATLAB will not be able to run that function until you clear the variable from memory.

Multiple Implementation Types

There are five file precedence types. MATLAB uses file precedence to select between identically named functions in the same directory. The order of precedence for file types is

1. Built-in file
2. MEX-files
3. MDL (Simulink® model) file
4. P-code file
5. M-file

For example, if MATLAB finds a P-code and an M-file version of a method in a class directory, then the P-code version is used. It is, therefore, important to regenerate the P-code version whenever you edit the M-file.

Querying Which Function Gets Called

You can determine which function MATLAB will call using the `which` command. For example,

```
which pie3
matlabroot/toolbox/matlab/specgraph/pie3.m
```

However, if `p` is a portfolio object,

```
which pie3(p)
dir_on_your_path/@portfolio/pie3.m      % portfolio method
```

The `which` command determines which version of `pie3` MATLAB will call if you passed a portfolio object as the input argument. To see a list of all versions of a particular function that are on your MATLAB path, use the `-all` option. See the `which` reference page for more information on this command.

Calling External Functions

The MATLAB external interface offers a number of ways to run external functions from MATLAB. This includes programs written in C or Fortran,

methods invoked on Sun Java or COM (Component Object Model) objects, functions that interface with serial port hardware, and functions stored in shared libraries. The *MATLAB External Interfaces* documentation describes these various interfaces and how to call these external functions.

Running External Programs

For information on how to invoke operating systems commands or execute programs that are external to MATLAB, see Running External Programs in the MATLAB Desktop Tools and Development documentation.

Function Arguments

In this section...

Overview

When calling a function, the caller provides the function with any data it needs by passing the data in an argument list. Data that needs to be returned to the caller is passed back in a list of return values.

Semantically speaking, the MATLAB software always passes argument data by value. (Internally, MATLAB optimizes away any unnecessary copy operations.)

If you pass data to a function that then modifies this data, you will need to update your own copy of the data. You can do this by having the function return the updated value as an output argument.

Passing Certain Argument Types

This section explains how to pass the following types of data in a function call:

- "Passing Strings" on page 3-40
- "Passing Filenames" on page 3-40
- "Passing Function Handles" on page 3-41

Passing Strings

When using the function syntax to pass a string literal to a function, you must enclose the string in single quotes, (`'string'`). For example, to create a new directory called `myapptests`, use

```
mkdir('myapptests')
```

On the other hand, variables that contain strings do not need to be enclosed in quotes:

```
dirname = 'myapptests';
mkdir(dirname)
```

Passing Filenames

You can specify a filename argument using the MATLAB command or function syntax. For example, either of the following are acceptable. (The `.mat` file extension is optional for `save` and `load`):

```
load mydata.mat          % Command syntax
load('mydata.mat')       % Function syntax
```

If you assign the output to a variable, you must use the function syntax:

```
savedData = load('mydata.mat')
```

Specify ASCII files as shown here. In this case, the file extension is required:

```
load mydata.dat -ascii            % Command syntax
load('mydata.dat','-ascii')       % Function syntax
```

Determining Filenames at Run-Time. There are several ways that your function code can work on specific files without your having to hardcode their filenames into the program. You can

- Pass the filename as an argument:

  ```
  function myfun(datafile)
  ```

- Prompt for the filename using the `input` function:

  ```
  filename = input('Enter name of file:  ', 's');
  ```

- Browse for the file using the `uigetfile` function:

```
[filename, pathname] = uigetfile('*.mat', 'Select MAT-file');
```

Passing Function Handles

The MATLAB function handle has several uses, the most common being a means of immediate access to the function it represents. You can pass function handles in argument lists to other functions, enabling the receiving function to make calls by means of the handle.

To pass a function handle, include its variable name in the argument list of the call:

```
fhandle = @humps;
x = fminbnd(fhandle, 0.3, 1);
```

The receiving function invokes the function being passed using the usual MATLAB calling syntax:

```
function [xf, fval, exitflag, output] = ...
         fminbnd(fhandle, ax, bx, options, varargin)
            .
            .
            .
113  fx = fhandle(x, varargin{:});
```

Passing Arguments in Structures or Cell Arrays

Instead of requiring an additional argument for every value you want to pass in a function call, you can package them in a MATLAB structure or cell array.

Passing Arguments in a Structure

Make each input you want to pass a separate field in the structure argument, using descriptive names for the fields. Structures allow you to change the number, contents, or order of the arguments without having to modify the function. They can also be useful when you have a number of functions that need similar information.

This example updates weather statistics from information in the following chart.

City	Temp.	Heat Index	Wind Speed	Wind Chill
Boston	43	32	8	37
Chicago	34	27	3	30
Lincoln	25	17	11	16
Denver	15	-5	9	0
Las Vegas	31	22	4	35
San Francisco	52	47	18	42

The information is stored in structure `W`. The structure has one field for each column of data:

```
W = struct('city', {'Bos','Chi','Lin','Dnv','Vgs','SFr'}, ...
           'temp', {43, 34, 25, 15, 31, 52}, ...
           'heatix', {32, 27, 17, -5, 22, 47}, ...
           'wspeed', {8, 3, 11, 9, 4, 18}, ...
           'wchill', {37, 30, 16, 0, 35, 42});
```

To update the data base, you can pass the entire structure, or just one field with its associated data. In the call shown here, `W.wchill` is a comma-separated list:

```
updateStats(W.wchill);
```

Passing Arguments in a Cell Array

You can also group arguments into cell arrays. The advantage over structures is that cell arrays are referenced by index, allowing you to loop through a cell array and access each argument passed in or out of the function. The disadvantage is that you don't have field names to describe each variable.

This example passes several attribute-value arguments to the `plot` function:

```
X = -pi:pi/10:pi;
Y = tan(sin(X)) - sin(tan(X));
```

```
C{1,1} = 'LineWidth';          C{2,1} = 2;
C{1,2} = 'MarkerEdgeColor';    C{2,2} = 'k';
C{1,3} = 'MarkerFaceColor';    C{2,3} = 'g';

plot(X, Y, '--rs', C{:})
```

Assigning Output Arguments

Use the syntax shown here to store any values that are returned by the function you are calling. To store one output, put the variable that is to hold that output to the left of the equal sign:

```
vout = myfun(vin1, vin2, ...);
```

To store more than one output, list the output variables inside square brackets and separate them with commas or spaces:

```
[vout1 vout2 ...] = myfun(vin1, vin2, ...);
```

The number of output variables in your function call statement does not have to match the number of return values declared in the function being called. For a function that declares N return values, you can specify anywhere from zero to N output variables in the call statement. Any return values that you do not have an output variable for are discarded.

Functions return output values in the order in which the corresponding output variables appear in the function definition line within the M-file. This function returns 100 first, then `x * y`, and lastly `x.^2`:

```
function [a b c] = myfun(x, y)
b = x * y;    a = 100;    c = x.^2;
```

If called with only one output variable in the call statement, the function returns only 100 and discards the values of `b` and `c`. If called with no outputs, the functions returns 100 in the MATLAB default variable `ans`.

Assigning Optional Return Values

The section "Passing Variable Numbers of Arguments" on page 3-47 describes the method of returning optional outputs in a cell array called `varargout`.

A function that uses varargout to return optional values has a function definition line that looks like one of the following:

```
function varargout = myfun(vin1, vin2, ...)
function [vout1 vout2 ... varargout] = myfun(vin1, vin2, ...)
```

The code within the function builds the varargout cell array. The content and order of elements in the cell array determines how MATLAB assigns optional return values to output variables in the function call.

In the case where varargout is the only variable shown to the left of the equal sign in the function definition line, MATLAB assigns varargout{1} to the first output variable, varargout{2} to the second, and so on. If there are other outputs declared in the function definition line, then MATLAB assigns those outputs to the leftmost output variables in the call statement, and then assigns outputs taken from the varargout array to the remaining output variables in the order just described.

This function builds the varargout array using descending rows of a 5-by-5 matrix. The function is capable of returning up to six outputs:

```
function varargout = byRow(a)
varargout{1} = '    With VARARGOUT constructed by row ...';
for k = 1:5
    row = 5 - (k-1);              % Reverse row order
    varargout{k+1} = a(row,:);
end
```

Call the function, assigning outputs to four variables. MATLAB returns varargout{1:4}, with rows of the matrix in varargout{2:4} and in the order in which they were stored by the function:

```
[text r1 r2 r3] = byRow(magic(5))
text =
    With VARARGOUT constructed by row ...
r1 =
    11    18    25     2     9
r2 =
    10    12    19    21     3
r3 =
     4     6    13    20    22
```

A similar function builds the `varargout` array using diagonals of a 5-by-5 matrix:

```
function varargout = byDiag(a)
varargout{1} = '    With VARARGOUT constructed by diagonal ...';
for k = -4:4
    varargout{k + 6} = diag(a, k);
end
```

Call the function with five output variables. Again, MATLAB assigns elements of `varargout` according to the manner in which it was constructed within the function:

```
[text d1 d2 d3 d4] = byDiag(magic(5))
text =
    With VARARGOUT constructed by diagonal ...
d1 =
    11
d2 =
    10
    18
d3 =
     4
    12
    25
d4 =
    23
     6
    19
     2
```

Checking the Number of Input Arguments

The `nargin` and `nargout` functions enable you to determine how many input and output arguments a function is called with. You can then use conditional statements to perform different tasks depending on the number of arguments. For example,

```
function c = testarg1(a, b)
if (nargin == 1)
    c = a .^ 2;
```

```
elseif (nargin == 2)
    c = a + b;
end
```

Given a single input argument, this function squares the input value. Given two inputs, it adds them together.

Here is a more advanced example that finds the first token in a character string. A *token* is a set of characters delimited by white space or some other character. Given one input, the function assumes a default delimiter of white space; given two, it lets you specify another delimiter if desired. It also allows for two possible output argument lists:

```
function [token, remainder] = strtok(string, delimiters)
% Function requires at least one input argument
if nargin < 1
   error('Not enough input arguments.');
end
token = []; remainder = [];
len = length(string);
if len == 0
   return
end

% If one input, use white space delimiter
if (nargin == 1)
   delimiters = [9:13 32]; % White space characters
end
i = 1;

% Determine where nondelimiter characters begin
while (any(string(i) == delimiters))
   i = i + 1;
   if (i > len), return, end
end

% Find where token ends
start = i;
while (~any(string(i) == delimiters))
   i = i + 1;
```

```
      if (i > len), break, end
   end
   finish = i - 1;
   token = string(start:finish);

   % For two output arguments, count characters after
   % first delimiter (remainder)
   if (nargout == 2)
      remainder = string(finish+1:end);
   end
```

The `strtok` function is a MATLAB M-file in the `strfun` directory.

Note The order in which output arguments appear in the function declaration line is important. The argument that the function returns in most cases appears first in the list. Additional, optional arguments are appended to the list.

Passing Variable Numbers of Arguments

The `varargin` and `varargout` functions let you pass any number of inputs or return any number of outputs to a function. This section describes how to use these functions and also covers

- "Unpacking varargin Contents" on page 3-48
- "Packing varargout Contents" on page 3-48
- "varargin and varargout in Argument Lists" on page 3-49

MATLAB packs all specified input arguments into a *cell array*, a special kind of MATLAB array that consists of cells instead of array elements. Each cell can hold any size or kind of data — one might hold a vector of numeric data, another in the same array might hold an array of string data, and so on. For output arguments, your function code must pack them into a cell array so that MATLAB can return the arguments to the caller.

Here is an example function that accepts any number of two-element vectors and draws a line to connect them:

```
function testvar(varargin)
for k = 1:length(varargin)
   x(k) = varargin{k}(1); % Cell array indexing
   y(k) = varargin{k}(2);
end
xmin = min(0,min(x));
ymin = min(0,min(y));
axis([xmin fix(max(x))+3 ymin fix(max(y))+3])
plot(x,y)
```

Coded this way, the `testvar` function works with various input lists; for example,

```
testvar([2 3],[1 5],[4 8],[6 5],[4 2],[2 3])
testvar([-1 0],[3 -5],[4 2],[1 1])
```

Unpacking varargin Contents

Because `varargin` contains all the input arguments in a cell array, it's necessary to use cell array indexing to extract the data. For example,

```
y(n) = varargin{n}(2);
```

Cell array indexing has two subscript components:

- The indices within curly braces `{}` specify which cell to get the contents of.
- The indices within parentheses `()` specify a particular element of that cell.

In the preceding code, the indexing expression `{i}` accesses the `n`th cell of `varargin`. The expression `(2)` represents the second element of the cell contents.

Packing varargout Contents

When allowing a variable number of output arguments, you must pack all of the output into the `varargout` cell array. Use `nargout` to determine how many output arguments the function is called with. For example, this code accepts a two-column input array, where the first column represents a set of `x` coordinates and the second represents `y` coordinates. It breaks the array into

separate `[xi yi]` vectors that you can pass into the `testvar` function shown at the beginning of the section on "Passing Variable Numbers of Arguments" on page 3-47:

```
function [varargout] = testvar2(arrayin)
for k = 1:nargout
   varargout{k} = arrayin(k,:);   % Cell array assignment
end
```

The assignment statement inside the `for` loop uses cell array assignment syntax. The left side of the statement, the cell array, is indexed using curly braces to indicate that the data goes inside a cell. For complete information on cell array assignment, see the documentation on Cell Arrays

To call `testvar2`, type

```
a = [1 2; 3 4; 5 6; 7 8; 9 0];

[p1, p2, p3, p4, p5] = testvar2(a)
p1 =
     1     2
p2 =
     3     4
p3 =
     5     6
p4 =
     7     8
p5 =
     9     0
```

varargin and varargout in Argument Lists

`varargin` or `varargout` must appear last in the argument list, following any required input or output variables. That is, the function call must specify the required arguments first. For example, these function declaration lines show the correct placement of `varargin` and `varargout`:

```
function [out1,out2] = example1(a,b,varargin)
function [i,j,varargout] = example2(x1,y1,x2,y2,flag)
```

Parsing Inputs with inputParser

MATLAB provides a class called `inputParser` to handle the different types of arguments passed into an M-file function. Using `inputParser`, you create a schema that both represents and verifies the content of the entire list of input arguments passed on a call to the function. When used in all of your code development, this schema offers a consistent and thorough means of managing and validating the input information.

This section covers the following topics

To illustrate how to use the inputParser class, the documentation in this section develops a new M-file program called `publish_ip`, (based on the MATLAB `publish` function). There are three calling syntaxes for this function:

```
publish_ip('scriptfile')
publish_ip('scriptfile', 'format')
publish_ip('scriptfile', options)
```

There is one required argument (`scriptfile`), one optional argument (`format`), and a number of optional arguments that are specified as parameter-value pairs (`options`).

Defining a Specification for Each Input Parameter

Most programs have a block of code toward the beginning that parses the values in the input argument list and checks these values against what is expected. The `inputParser` class provides the following methods with which you can specify what the inputs are and whether they are required, optional, or to be specified using the parameter-value syntax:

- `addRequired` — Add a required parameter to the schema
- `addOptional` — Add an optional parameter to the schema
- `addParamValue` — Add an optional parameter-value pair to the schema

Creating the inputParser Object. Call the class constructor for `inputParser` to create an instance of the class. This class instance, or object, gives you access to all of the methods and properties of the class.

Begin writing the example `publish_ip` M-file by entering the following two statements:

```
function x = publish_ip(scriptfile, varargin)
p = inputParser;   % Create an instance of the class.
```

After calling the constructor, use the `addRequired`, `addOptional`, and `addParamValue` methods to add arguments to the schema.

Note The constructor and all methods and properties of the `inputParser` class are case sensitive.

Adding Arguments to the Schema. Add any required arguments to the schema using the `addRequired` method. This method takes two inputs: the name of the required parameter, and an optional handle to a function that validates the parameter:

```
addRequired(name, validator);
```

Put an `addRequired` statement at the end of your `publish_ip` code. The two arguments for `addRequired` in this case are a parameter name `script` to represent the filename input, and a handle to a function that will validate the

filename, `ischar`. After adding the `addRequired` statement, your `publish_ip` function should now look like this:

```
function x = publish_ip(scriptfile, varargin)
p = inputParser;   % Create an instance of the class.

p.addRequired('script', @ischar);
```

Use the `addOptional` method to add any arguments that are not required. The syntax for `addOptional` is similar to that of `addRequired` except that you also need to specify a default value to be used whenever the optional argument is not passed:

```
addOptional(name, default, validator);
```

Add the following statement to your `publish_ip` M-file. In this case, the `validator` input is a handle to an anonymous function:

```
p.addOptional('format', 'html', ...
    @(x)any(strcmpi(x,{'html','ppt','xml','latex'})));
```

Use `addParamValue` to specify any arguments that use a parameter-value format. The syntax is

```
addParamValue(name, default, validator);
```

Add the following code to your `publish_ip` M-file:

```
p.addParamValue('outputDir', pwd, @ischar);
p.addParamValue('maxHeight', [], @(x)x>0 && mod(x,1)==0);
p.addParamValue('maxWidth', [], @(x)x>0 && mod(x,1)==0);
```

Listing the Parameters. At this point, the schema is complete. Here is the file `publish_ip.m`:

```
function x = publish_ip(scriptfile, varargin)
p = inputParser;   % Create an instance of the class.

p.addRequired('script', @ischar);

p.addOptional('format', 'html', ...
    @(x)any(strcmpi(x,{'html','ppt','xml','latex'})));
```

```
p.addParamValue('outputDir', pwd, @ischar);
p.addParamValue('maxHeight', [], @(x)x>0 && mod(x,1)==0);
p.addParamValue('maxWidth', [], @(x)x>0 && mod(x,1)==0);
```

When you call the program, MATLAB stores the name of each argument in the `Parameters` property of object `p`. Add the following statement to your `publish_ip` M-file to display `p.Parameters`:

```
disp 'The input parameters for this program are'
disp(p.Parameters)
```

Save the M-file, and then run it as shown here:

```
publish_ip('ipscript.m', 'ppt', 'outputDir', ...
    'C:/matlab/test', 'maxWidth', 500, 'maxHeight', 300);
```

The output is

```
The input parameters for this program are
   'format'   'maxHeight'   'maxWidth'   'outputDir'   'script'
```

Parsing Parameter Values on the Function Call

Once you have constructed a schema that represents all possible inputs to the function, the next task is to write the code that parses and verifies these arguments whenever the function is called. The `parse` method of the `inputParser` class reads and validates the required `scriptfile` argument and any optional arguments that follow it in the argument list:

```
p.parse(scriptfile, varargin{:});
```

Execution of the parse method validates each argument and also builds a structure from the input arguments. The name of the structure is `Results`, which is accessible as a property of the object. To get the value of all arguments, type

```
p.Results
```

To get the value of any single input argument, type

```
p.Results.argname
```

where `argname` is the name of the argument. Continue with the `publish_ip` exercise started earlier in this section by removing the two `disp` statements that were inserted in the last section, and then adding the following lines:

```
% Parse and validate all input arguments.
p.parse(scriptfile, varargin{:});

% Display the value of a specific argument.
disp' '
fprintf('\nThe maximum height is %d.\n', ...
     p.Results.maxHeight)

% Display all arguments.
disp ' '
disp 'List of all arguments:'
disp(p.Results)
```

Now save and execute the M-file, passing the required script file argument, the optional format argument, as well as several parameter-value arguments. MATLAB assigns those values you pass in the argument list to the appropriate fields of the `Results` structure:

```
publish_ip('ipscript.m', 'ppt', 'outputDir', ...
    'C:/matlab/test', 'maxWidth', 500, 'maxHeight', 300);

The maximum height is 300.

List of all arguments:
              format: 'ppt'
           maxHeight: 300
            maxWidth: 500
           outputDir: 'C:/matlab/test'
              script: 'ipscript.m'
```

Packaging Arguments in a Structure

By setting the `StructExpand` property of the `inputParser` object to `true`, you can pass arguments to your function in the form of a structure instead of

individually in the argument list. This property must be set prior to calling the `parse` method.

`StructExpand` defaults to the `true` state, so you don't have to make any changes to your test program to run this example.

Put all of the input arguments into a structure:

```
s.format = 'xml';
s.outputDir = 'C:/matlab/test';
s.maxWidth = 200;
s.maxHeight = 150;
```

Now call the function, passing the filename and input structure:

```
publish_ip('ipscript.m', s);

The maximum height is 150.

List of all arguments:
          format: 'xml'
       maxHeight: 150
        maxWidth: 200
       outputDir: 'C:/matlab/test'
          script: 'ipscript.m'
```

To disable struct expansion, you can include the following statement somewhere in your program code before the `p.parse` statement:

```
p.StructExpand = false;
```

For this example however, leave the `StructExpand` in its default `true` state.

Overriding the Input Structure. If you want to pass your argument list in a structure, as described in the previous section, but you also want to alter the value of one or more of these arguments without having to modify the structure, you can do so by passing both the structure and the modified argument:

```
publish_ip('ipscript.m', s, ...
           'outputDir', 'C:/matlab/R2007a/temp');
```

```
List of all arguments:
              format: 'xml'
           maxHeight: 150
            maxWidth: 200
           outputDir: 'C:/matlab/R2007a/temp'
              script: 'ipscript.m'
```

Arguments That Default

Any arguments that you do not include in a call to your function are given their default values by MATLAB. You defined these default values when you created your schema using the `addOptional` and `addParamValue` methods. The `UsingDefaults` property is a cell array that contains the names of any arguments that were not passed in the function call, and thus were assigned default values.

In your example M-file, remove or comment out the lines that display the maximum height and the list of all arguments, and add the following lines in their place:

```
% Show which arguments were not specified in the call.
disp' '
disp 'List of arguments given default values:'
for k=1:numel(p.UsingDefaults)
   field = char(p.UsingDefaults(k));
   value = p.Results.(field);
   if isempty(value),   value = '[]';   end
   disp(sprintf('   ''%s''    defaults to %s', field, value))
end
```

Save the M-file and run it without specifying the `format`, `outputDir`, or `maxHeight` arguments:

```
publish_ip('ipscript.m', 'maxWidth', 500);

List of arguments given default values:
   'format'    defaults to html
   'outputDir'    defaults to D:\matlabtest
   'maxHeight'    defaults to []
```

Note that `outputDir` defaults to your current working directory, as specified near the beginning of your example M-file in the statement

```
p.addParamValue('outputDir', pwd, @ischar);
```

Validating the Input Arguments

When you call your function, MATLAB checks any arguments for which you have specified a validator function. If the validator finds an error, MATLAB displays an error message and aborts the function. In the `publish` function example, the `outputDir` argument validates the value passed in using `@ischar`.

Pass a number instead of a string for the `outputDir` argument:

```
publish_ip('ipscript.m', 'outputDir', 5);
??? Error using ==> publish_ip at 14
Argument 'outputDir' failed validation ischar.
```

Handling Unmatched Arguments. MATLAB throws an error if you call your function with any arguments that are not part of the `inputParser` schema. You can disable this error by setting the `KeepUnmatched` property to `true`. When `KeepUnmatched` is in the `true` state, MATLAB does not throw an error, but instead stores any arguments that are not in the schema in a cell array of strings accessible through the `Unmatched` property of the object. `KeepUnmatched` defaults to `false`.

In your `publish_ip` M-file, remove or comment out all of the code that follows the `p.addParamValue` statements, and add the following code in its place:

```
p.KeepUnmatched = true;

% Parse and validate all input arguments.
p.parse(scriptfile, varargin{:});

disp ' '
disp 'List of unmatched arguments:'
disp(p.Unmatched)
```

Save and run the function, passing two arguments that are not defined in the schema:

```
publish_ip('ipscript.m', s, ...
           'outputDir', 'C:/matlab/R2007a/temp', ...
           'colorSpace', 'CMYK', 'density', 200);

List of unmatched arguments:
    colorSpace: 'CMYK'
       density: 200
```

Enabling Case-Sensitive Matching. When you pass optional arguments in the function call, MATLAB compares these arguments with the names of parameter-value argument names in the schema. By default, MATLAB does not use case sensitivity in this comparison. So, an argument name entered into the schema (using `addParamValue`) as `maxHeight` will match an argument passed as `MAXHeight` in the function call. You can override the default and make these comparisons case sensitive by setting the `CaseSensitive` property of the object to `true`. MATLAB does not error on a case mismatch, however, unless the `KeepUnmatched` property is set to `false`: its default state.

At some point in your `publish_ip` M-file before executing the `parse` method, set `KeepUnmatched` to `false` and `CaseSensitive` to `true`. You can also remove the statements that display the list of unmatched arguments. Now execute the `publish_ip` function using `MAXHeight` as the name of the argument for specifying maximum height:

```
p.KeepUnmatched = false;
p.CaseSensitive = true;

% Parse and validate all input arguments.
p.parse(scriptfile, varargin{:});
```

Save and run the function, using `MAXHeight` as the name of the argument for specifying maximum height:

```
publish_ip('ipscript.m', 'ppt', 'outputDir', ...
    'C:/matlab/test', 'maxWidth', 500, 'MAXHeight', 300);
??? Error using ==> publish_ip at 17
Argument 'MaxHeight' did not match any valid parameter of the parser.
```

Adding the Function Name to Error Messages. Use the `FunctionName` property to include the name of your function in error messages thrown by the validating function:

At some point in your `publish_ip` M-file before executing the `parse` method, set the `FunctionName` property to `PUBLISH_IP`, and then run the function:

```
p.FunctionName = 'PUBLISH_IP';

% Parse and validate all input arguments.
p.parse(scriptfile, varargin{:});
```

Save and run the function and observe text of the error message:

```
publish_ip('ipscript.m', 'ppt', 'outputDir', 5, ...
    'maxWidth', 500, 'maxHeight', 300);
??? Error using ==> PUBLISH_IP
Argument 'outputDir' failed validation ischar.
```

Making a Copy of the Schema

The `createCopy` method enables you to make a copy of an existing schema. Because the `inputParser` class uses handle semantics, you cannot make a copy of the object using an assignment statement.

The following statement creates an `inputParser` object `s` that is a copy of `p`:

```
s = p.createCopy
```

Summary of inputParser Methods

Method	Description
`addOptional`	Add an optional argument to the schema
`addParamValue`	Add a parameter-value pair argument to the schema
`addRequired`	Add a required argument to the schema

Method	Description
createCopy	Create a copy of the inputParser object
parse	Parse and validate the named inputs

Summary of inputParser Properties that Control Parsing

Property	Description
CaseSensitivity	Enable or disable case-sensitive matching of argument names. Defaults to false.
FunctionName	Function name to be included in error messages. Defaults to an empty string.
KeepUnmatched	Enable or disable errors on unmatched arguments. Defaults to false.
StructExpand	Enable or disable passing arguments in a structure. Defaults to true.

Summary of inputParser Properties that Provide Information

Property	Description
Parameters	Names of arguments defined in inputParser schema.
Results	Names and values of arguments passed in function call that are in the schema for this function.
Unmatched	Names and values of arguments passed in function call that are not in the schema for this function.
UsingDefaults	Names of arguments not passed in function call that are given default values.

Passing Optional Arguments to Nested Functions

You can use optional input and output arguments with nested functions, but you should be aware of how MATLAB interprets `varargin`, `varargout`, `nargin`, and `nargout` under those circumstances.

`varargin` and `varargout` are variables and, as such, they follow exactly the same scoping rules as any other MATLAB variable. Because nested functions share the workspaces of all outer functions, `varargin` and `varargout` used in a nested function can refer to optional arguments passed to or from the nested function, or passed to or from one of its outer functions.

`nargin` and `nargout`, on the other hand, are functions and when called within a nested function, always return the number of arguments passed to or from the nested function itself.

Using varargin and varargout

`varargin` or `varargout` used in a nested function can refer to optional arguments passed to or from that function, or to optional arguments passed to or from an outer function.

- If a nested function includes `varargin` or `varargout` in its function declaration line, then the use of `varargin` or `varargout` within that function returns optional arguments passed to or from that function.
- If `varargin` or `varargout` are not in the nested function declaration but are in the declaration of an outer function, then the use of `varargin` or `varargout` within the nested function returns optional arguments passed to the outer function.

In the example below, function `C` is nested within function `B`, and function `B` is nested within function `A`. The term `varargin{1}` in function `B` refers to the second input passed to the primary function `A`, while `varargin{1}` in function `C` refers to the first argument, `z`, passed from function `B`:

```
function x = A(y, varargin)   % Primary function A
B(nargin, y * rand(4))

   function B(argsIn, z)      % Nested function B
   if argsIn >= 2
      C(z, varargin{1}, 4.512, 1.729)
```

```
      end

         function C(varargin)    % Nested function C
         if nargin >= 2
            x = varargin{1}
         end
         end   % End nested function C
      end   % End nested function B
   end   % End primary function A
```

Using nargin and nargout

When nargin or nargout appears in a nested function, it refers to the number of inputs or outputs passed to that particular function, regardless of whether or not it is nested.

In the example shown above, nargin in function A is the number of inputs passed to A, and nargin in function C is the number of inputs passed to C. If a nested function needs the value of nargin or nargout from an outer function, you can pass this value in as a separate argument, as done in function B.

Example of Passing Optional Arguments to Nested Functions

This example references the primary function's varargin cell array from each of two nested functions. (Because the workspace of an outer function is shared with all functions nested within it, there is no need to pass varargin to the nested functions.)

Both nested functions make use of the nargin value that applies to the primary function. Calling nargin from the nested function would return the number of inputs passed to that nested function, and not those that had been passed to the primary. For this reason, the primary function must pass its nargin value to the nested functions.

```
function meters = convert2meters(miles, varargin)
% Converts MILES (plus optional FEET and INCHES input)
% values to METERS.

if nargin < 1 || nargin > 3
   error('1 to 3 input arguments are required');
```

```
end

   function feet = convert2Feet(argsIn)
   % Nested function that converts miles to feet and adds in
   % optional FEET argument.

   feet = miles .* 5280;

   if argsIn >= 2
      feet = feet + varargin{1};
   end
   end  % End nested function convert2Feet

   function inches = convert2Inches(argsIn)
   % Nested function that converts feet to inches and adds in
   % optional INCHES argument.

   inches = feet .* 12;

   if argsIn == 3
      inches = inches + varargin{2};
   end
   end  % End nested function convert2Inches

feet = convert2Feet(nargin);
inches = convert2Inches(nargin);

meters = inches .* 2.54 ./ 100;
end  % End primary function convert2meters

convert2meters(5)
ans =
  8.0467e+003

convert2meters(5, 2000, 4.7)
ans =
  8.6564e+003
```

Returning Modified Input Arguments

If you pass any input variables that the function can modify, you will need to include the same variables as output arguments so that the caller receives the updated value.

For example, if the function `readText`, shown below, reads one line of a file each time is it called, then it must keep track of the offset into the file. But when `readText` terminates, its copy of the `offset` variable is cleared from memory. To keep the offset value from being lost, `readText` must return this value to the caller:

```
function [text, offset] = readText(filestart, offset)
```

4

Types of Functions

Overview of MATLAB Function Types

There are essentially two ways to create a new function for your MATLAB application: in a command entered at run-time, or in a file saved to permanent storage.

The command-oriented function, called an *anonymous function*, is relatively brief in its content. It consists of a single MATLAB statement that can interact with multiple input and output arguments. The benefit of using anonymous functions is that you do not have to edit and maintain a file for functions that require only a brief definition.

There are several types of functions that are stored in files (called M-files). The most basic of these are *primary functions* and *subfunctions*. Primary functions are visible to other functions outside of their M-file, while subfunctions, generally speaking, are not. That is, you can call a primary function from an anonymous function or from a function defined in a separate M-file, whereas you can call a subfunction only from functions within the same M-file. (See the Description section of the `function_handle` reference page for information on making a subfunction externally visible.)

Two specific types of primary M-file functions are the *private* and *overloaded function*. Private functions are visible only to a limited group of other functions. This type of function can be useful if you want to limit access to a function, or when you choose not to expose the implementation of a function. Overloaded functions act the same way as overloaded functions in most computer languages. You can create multiple implementations of a function so that each responds accordingly to different types of inputs.

The last type of MATLAB function is the *nested function*. Nested functions are not an independent function type; they exist within the body of one of the other types of functions discussed here (with the exception of anonymous functions), and also within other nested functions.

Anonymous Functions

In this section...

Constructing an Anonymous Function

Anonymous functions give you a quick means of creating simple functions without having to create M-files each time. You can construct an anonymous function either at the MATLAB command line or in any M-file function or script.

The syntax for creating an anonymous function from an expression is

```
fhandle = @(arglist) expr
```

Starting from the right of this syntax statement, the term `expr` represents the body of the function: the code that performs the main task your function is to accomplish. This consists of any single, valid MATLAB expression. Next is `arglist`, which is a comma-separated list of input arguments to be passed to the function. These two components are similar to the body and argument list components of any function.

Leading off the entire right side of this statement is an `@` sign. The `@` sign is the MATLAB operator that constructs a function handle. Creating a function handle for an anonymous function gives you a means of invoking the function. It is also useful when you want to pass your anonymous function in a call to some other function. The `@` sign is a required part of an anonymous function definition.

Note Function handles not only provide access to anonymous functions. You can create a function handle to any MATLAB function. The constructor uses a different syntax: `fhandle = @functionname` (e.g., `fhandle = @sin`). To find out more about function handles, see "Function Handles" on page 1-126 in the Programming Fundamentals documentation.

The syntax statement shown above constructs the anonymous function, returns a handle to this function, and stores the value of the handle in variable `fhandle`. You can use this function handle in the same way as any other MATLAB function handle.

Simple Example

The statement below creates an anonymous function that finds the square of a number. When you call this function, MATLAB assigns the value you pass in to variable `x`, and then uses `x` in the equation `x.^2`:

```
sqr = @(x) x.^2;
```

The `@` operator constructs a function handle for this function, and assigns the handle to the output variable `sqr`. As with any function handle, you execute the function associated with it by specifying the variable that contains the handle, followed by a comma-separated argument list in parentheses. The syntax is

```
fhandle(arg1, arg2, ..., argN)
```

To execute the `sqr` function defined above, type

```
a = sqr(5)
a =
   25
```

Because `sqr` is a function handle, you can pass it in an argument list to other functions. The code shown here passes the `sqr` anonymous function to the MATLAB `quad` function to compute its integral from zero to one:

```
quad(sqr, 0, 1)
ans =
```

```
    0.3333
```

A Two-Input Example

As another example, you could create the following anonymous function that uses two input arguments, `x` and `y`. (The example assumes that variables `A` and `B` are already defined):

```
sumAxBy = @(x, y) (A*x + B*y);

whos sumAxBy
Name        Size                    Bytes  Class

sumAxBy     1x1                        16  function_handle
```

To call this function, assigning `5` to `x` and `7` to `y`, type

```
sumAxBy(5, 7)
```

Evaluating With No Input Arguments

For anonymous functions that do not take any input arguments, construct the function using empty parentheses for the input argument list:

```
t = @() datestr(now);
```

Also use empty parentheses when invoking the function:

```
t()

ans =
04-Sep-2003 10:17:59
```

You must include the parentheses. If you type the function handle name with no parentheses, MATLAB just identifies the handle; it does not execute the related function:

```
t

t =
    @() datestr(now)
```

Arrays of Anonymous Functions

To store multiple anonymous functions in an array, use a cell array. The example shown here stores three simple anonymous functions in cell array A:

```
A = {@(x)x.^2, @(y)y+10, @(x,y)x.^2+y+10}
A =
    [@(x)x.^2]    [@(y)y+10]    [@(x,y)x.^2+y+10]
```

Execute the first two functions in the cell array by referring to them with the usual cell array syntax, A{1} and A{2}:

```
A{1}(4) + A{2}(7)
ans =
    33
```

Do the same with the third anonymous function that takes two input arguments:

```
A{3}(4, 7)
ans =
    33
```

Space Characters in Anonymous Function Elements

Note that while using space characters in the definition of any function can make your code easier to read, spaces in the body of an anonymous function that is defined in a cell array can sometimes be ambiguous to MATLAB. To ensure accurate interpretation of anonymous functions in cell arrays, you can do any of the following:

- Remove all spaces from at least the body (not necessarily the argument list) of each anonymous function:

  ```
  A = {@(x)x.^2, @(y)y+10, @(x, y)x.^2+y+10};
  ```

- Enclose in parentheses any anonymous functions that include spaces:

  ```
  A = {(@(x)x .^ 2), (@(y) y +10), (@(x, y) x.^2 + y+10)};
  ```

- Assign each anonymous function to a variable, and use these variable names in creating the cell array:

```
A1 = @(x)x .^ 2;  A2 = @(y) y +10;  A3 = @(x, y)x.^2 + y+10;
A = {A1, A2, A3};
```

Outputs from Anonymous Functions

As with other MATLAB functions, the number of outputs returned by an anonymous function depends mainly on how many variables you specify to the left of the equals (=) sign when you call the function.

For example, consider an anonymous function `getPersInfo` that returns a person's address, home phone, business phone, and date of birth, in that order. To get someone's address, you can call the function specifying just one output:

```
address = getPersInfo(name);
```

To get more information, specify more outputs:

```
[address, homePhone, busPhone] = getPersInfo(name);
```

Of course, you cannot specify more outputs than the maximum number generated by the function, which is four in this case.

Example

The anonymous `getXLSData` function shown here calls the MATLAB `xlsread` function with a preset spreadsheet filename (`records.xls`) and a variable worksheet name (`worksheet`):

```
getXLSData = @(worksheet) xlsread('records.xls', worksheet);
```

The `records.xls` worksheet used in this example contains both numeric and text data. The numeric data is taken from instrument readings, and the text data describes the category that each numeric reading belongs to.

Because the MATLAB `xlsread` function is defined to return up to three values (numeric, text, and raw data), `getXLSData` can also return this same number of values, depending on how many output variables you specify to the left of the equals sign in the call. Call `getXLSData` a first time, specifying only a single (numeric) output, `dNum`:

```
dNum = getXLSData('Week 12');
```

Display the data that is returned using a `for` loop. You have to use generic names (`v1`, `v2`, `v3`) for the categories, due to the fact that the text of the real category names was not returned in the call:

```
for k = 1:length(dNum)
   disp(sprintf('%s    v1: %2.2f    v2: %d    v3: %d', ...
      datestr(clock, 'HH:MM'), dNum(k,1), dNum(k,2), ...
      dNum(k,3)));
   end
```

Here is the output from the first call:

```
12:55   v1: 78.42    v2: 32    v3: 37
13:41   v1: 69.73    v2: 27    v3: 30
14:26   v1: 77.65    v2: 17    v3: 16
15:10   v1: 68.19    v2: 22    v3: 35
```

Now try this again, but this time specifying two outputs, numeric (`dNum`) and text (`dTxt`):

```
[dNum, dTxt] = getXLSData('Week 12');

for k = 1:length(dNum)
   disp(sprintf('%s    %s: %2.2f    %s: %d    %s: %d', ...
      datestr(clock, 'HH:MM'), dTxt{1}, dNum(k,1), ...
      dTxt{2}, dNum(k,2), dTxt{3}, dNum(k,3)));
   end
```

This time, you can display the category names returned from the spreadsheet:

```
12:55    Temp: 78.42    HeatIndex: 32    WindChill: 37
13:41    Temp: 69.73    HeatIndex: 27    WindChill: 30
14:26    Temp: 77.65    HeatIndex: 17    WindChill: 16
15:10    Temp: 68.19    HeatIndex: 22    WindChill: 35
```

Variables Used in the Expression

Anonymous functions commonly include two types of variables:

- Variables specified in the argument list. These often vary with each function call.

- Variables specified in the body of the expression. MATLAB captures these variables and holds them constant throughout the lifetime of the function handle.

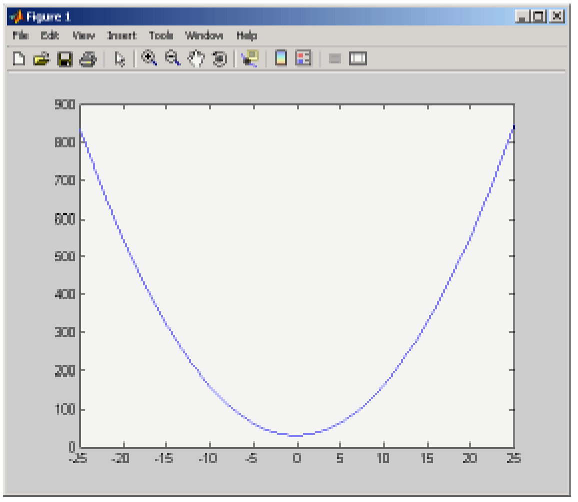
The latter variables must have a value assigned to them at the time you construct an anonymous function that uses them. Upon construction, MATLAB captures the current value for each variable specified in the body of that function. The function will continue to associate this value with the variable even if the value should change in the workspace or go out of scope.

The fact that MATLAB captures the values of these variables when the handle to the anonymous function is constructed enables you to execute an anonymous function from anywhere in the MATLAB environment, even outside the scope in which its variables were originally defined. But it also means that to supply new values for any variables specified within the expression, you must reconstruct the function handle.

Changing Variables Used in an Anonymous Function

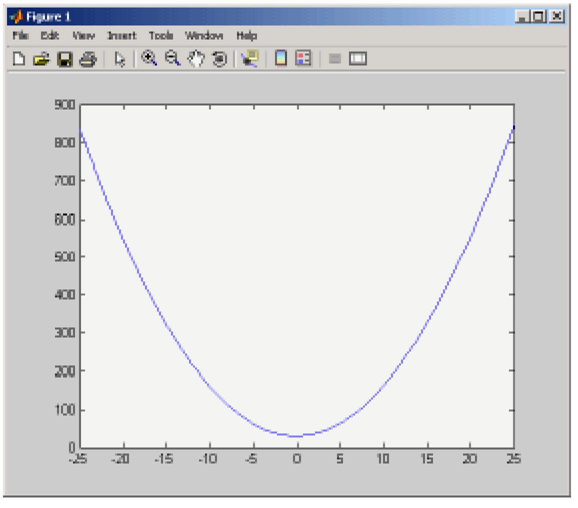
The second statement shown below constructs a function handle for an anonymous function called `parabola` that uses variables `a`, `b`, and `c` in the expression. Passing the function handle to the MATLAB `fplot` function plots it out using the initial values for these variables:

```
a = 1.3;   b = .2;   c = 30;
parabola = @(x) a*x.^2 + b*x + c;
fplot(parabola, [-25 25])
```



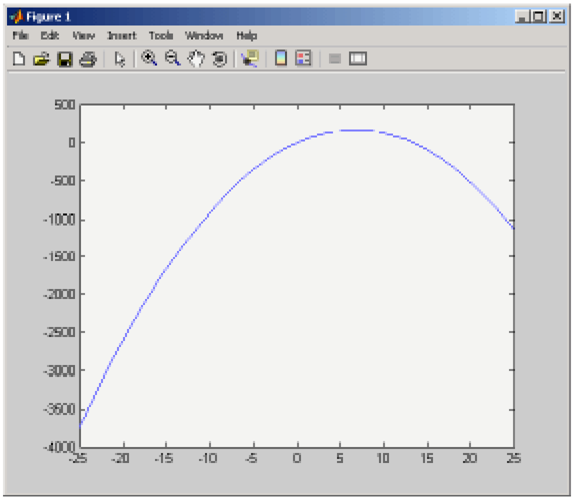
If you change the three variables in the workspace and replot the figure, the parabola remains unchanged because the `parabola` function is still using the initial values of `a`, `b`, and `c`:

```
a = -3.9;   b = 52;   c = 0;
fplot(parabola, [-25 25])
```



To get the function to use the new values, you need to reconstruct the function handle, causing MATLAB to capture the updated variables. Replot using the new construct, and this time the parabola takes on the new values:

```
a = -3.9;   b = 52;   c = 0;
parabola = @(x) a*x.^2 + b*x + c;
fplot(parabola, [-25 25])
```



For the purposes of this example, there is no need to store the handle to the anonymous function in a variable (`parabola`, in this case). You can just construct and pass the handle right within the call to `fplot`. In this way, you update the values of `a`, `b`, and `c` on each call:

```
fplot(@(x) a*x.^2 + b*x + c, [-25 25])
```

Examples of Anonymous Functions

This section shows a few examples of how you can use anonymous functions. These examples are intended to show you how to program with this type of function. For more mathematically oriented examples, see the MATLAB Mathematics documentation.

The examples in this section include

- "Example 1 — Passing a Function to quad" on page 4-12
- "Example 2 — Multiple Anonymous Functions" on page 4-13

Example 1 — Passing a Function to quad

The equation shown here has one variable `t` that can vary each time you call the function, and two additional variables, `g` and `omega`. Leaving these two variables flexible allows you to avoid having to hardcode values for them in the function definition:

```
x = g * cos(omega * t)
```

One way to program this equation is to write an M-file function, and then create a function handle for it so that you can pass the function to other functions, such as the MATLAB `quad` function as shown here. However, this requires creating and maintaining a new M-file for a purpose that is likely to be temporary, using a more complex calling syntax when calling `quad`, and passing the `g` and `omega` parameters on every call. Here is the function M-file:

```
function f = vOut(t, g, omega)
f = g * cos(omega * t);
```

This code has to specify `g` and `omega` on each call:

```
g = 2.5;  omega = 10;

quad(@vOut, 0, 7, [], [], g, omega)
ans =
   0.1935

quad(@vOut, -5, 5, [], [], g, omega)
ans =
   -0.1312
```

You can simplify this procedure by setting the values for `g` and `omega` just once at the start, constructing a function handle to an anonymous function that only lasts the duration of your MATLAB session, and using a simpler syntax when calling `quad`:

```
g = 2.5;  omega = 10;
```

```
quad(@(t) (g * cos(omega * t)), 0, 7)
ans =
   0.1935

quad(@(t) (g * cos(omega * t)), -5, 5)
ans =
   -0.1312
```

To preserve an anonymous function from one MATLAB session to the next, save the function handle to a MAT-file

```
save anon.mat f
```

and then load it into the MATLAB workspace in a later session:

```
load anon.mat f
```

Example 2 — Multiple Anonymous Functions

This example solves the following equation by combining two anonymous functions:

$$g(c) = \int_0^1 (x^2 + cx + 1)dx$$

The equivalent anonymous function for this expression is

```
g = @(c) (quad(@(x) (x.^2 + c*x + 1), 0, 1));
```

This was derived as follows. Take the parenthesized part of the equation (the integrand) and write it as an anonymous function. You don't need to assign the output to a variable as it will only be passed as input to the quad function:

```
@(x) (x.^2 + c*x + 1)
```

Next, evaluate this function from zero to one by passing the function handle, shown here as the entire anonymous function, to quad:

```
quad(@(x) (x.^2 + c*x + 1), 0, 1)
```

Supply the value for `c` by constructing an anonymous function for the entire equation and you are done:

```
g = @(c) (quad(@(x) (x.^2 + c*x + 1), 0, 1));

g(2)
ans =
   2.3333
```

Primary M-File Functions

The first function in any M-file is called the *primary function.* Following the primary function can be any number of subfunctions, which can serve as subroutines to the primary function.

Under most circumstances, the primary function is the only function in an M-file that you can call from the MATLAB command line or from another M-file function. You invoke this function using the name of the M-file in which it is defined.

For example, the `average` function shown here resides in the file `average.m`:

```
function y = average(x)
% AVERAGE Mean of vector elements.

y = sum(x)/length(x);      % Actual computation
```

You can invoke this function from the MATLAB command line with this command to find the average of three numbers:

```
average([12 60 42])
```

Note that it is customary to give the primary function the same name as the M-file in which it resides. If the function name differs from the filename, then you must use the filename to invoke the function.

Nested Functions

In this section...

Writing Nested Functions

You can define one or more functions within another function in your MATLAB application. These inner functions are said to be *nested* within the function that contains them. You can also nest functions within other nested functions. You cannot however define a nested function inside any of the MATLAB program control statements. This includes any block of code that is controlled by an `if`, `else`, `elseif`, `switch`, `for`, `while`, `try`, or `catch` statement.

To write a nested function, simply define one function within the body of another function in an M-file. Like any M-file function, a nested function contains any or all of the components described in "Basic Parts of an M-File" on page 3-8 in the Programming Fundamentals documentation. In addition, you must always terminate a nested function with an `end` statement:

```
function x = A(p1, p2)
...
   function y = B(p3)
   ...
   end
...
end
```

Note M-file functions don't normally require a terminating `end` statement. This rule does not hold, however, when you nest functions. If an M-file contains one or more nested functions, you must terminate *all* functions (including subfunctions) in the M-file with `end`, whether or not they contain nested functions.

Example — More Than One Nested Function

This example shows function `A` and two additional functions nested inside `A` at the same level:

```
function x = A(p1, p2)
...
   function y = B(p3)
   ...
   end

   function z = C(p4)
   ...
   end
...
end
```

Example — Multiply Nested Functions

This example shows multiply nested functions, `C` nested inside `B`, and `B` in `A`:

```
function x = A(p1, p2)
...
   function y = B(p3)
   ...
      function z = C(p4)
      ...
      end
   ...
   end
...
end
```

Calling Nested Functions

You can call a nested function

- From the level immediately above it. (In the following code, function A can call B or D, but not C or E.)
- From a function nested at the same level within the same parent function. (Function B can call D, and D can call B.)
- From a function at any lower level. (Function C can call B or D, but not E.)

```
function A(x, y)              % Primary function
B(x, y);
D(y);

   function B(x, y)           % Nested in A
   C(x);
   D(y);

      function C(x)           % Nested in B
      D(x);
      end
   end

   function D(x)              % Nested in A
   E(x);

      function E(x)           % Nested in D
      ...
      end
   end
end
```

You can also call a subfunction from any nested function in the same M-file.

You can pass variable numbers of arguments to and from nested functions, but you should be aware of how MATLAB interprets `varargin`, `varargout`, `nargin`, and `nargout` under those circumstances. See "Passing Optional Arguments to Nested Functions" in the MATLAB Programming Fundamentals documentation for more information on this.

Note If you construct a function handle for a nested function, you can call the nested function from any MATLAB function that has access to the handle. See "Using Function Handles with Nested Functions" on page 4-21.

Nested functions are not accessible to the `str2func` or `feval` function. You cannot call a nested function using a handle that has been constructed with `str2func`. And, you cannot call a nested function by evaluating the function name with `feval`. To call a nested function, you must either call it directly by name, or construct a function handle for it using the `@` operator.

Variable Scope in Nested Functions

The scope of a variable is the range of functions that have direct access to the variable to set, modify, or acquire its value. When you define a local (i.e., nonglobal) variable within a function, its scope is normally restricted to that function alone. For example, subfunctions do not share variables with the primary function or with other subfunctions. This is because each function and subfunction stores its variables in its own separate workspace.

Like other functions, a nested function has its own workspace. But it also has access to the workspaces of all functions in which it is nested. So, for example, a variable that has a value assigned to it by the primary function can be read or overwritten by a function nested at any level within the primary. Similarly, a variable that is assigned in a nested function can be read or overwritten by any of the functions containing that function.

In the following two examples, variable `x` is stored in the workspace of the outer `varScope` function and can be read or written to by all functions nested within it.

<pre>function varScope1 x = 5; nestfun1 function nestfun1 nestfun2 function nestfun2 x = x + 1 end end end</pre>	<pre>function varScope2 nestfun1 function nestfun1 nestfun2 function nestfun2 x = 5; end end x = x + 1 end</pre>

As a rule, a variable used or defined within a nested function resides in the workspace of the outermost function that both contains the nested function and accesses that variable. The scope of this variable is then the function to which this workspace belongs, and all functions nested to any level within that function.

In the next example, the outer function, `varScope3`, does not access variable `x`. Following the rule just stated, `x` is unknown to the outer function and thus is not shared between the two nested functions. In fact, there are two separate `x` variables in this example: one in the function workspace of `nestfun1` and one in the function workspace of `nestfun2`. When `nestfun2` attempts to update `x`, it fails because `x` does not yet exist in this workspace:

```
function varScope3
nestfun1
nestfun2

   function nestfun1
      x = 5;
   end

   function nestfun2
      x = x + 1
   end
end
```

The Scope of Output Variables

Variables containing values returned by a nested function are not in the scope of outer functions. In the two examples shown here, the one on the left fails in the second to last line because, although the *value* of `y` is returned by the nested function, the *variable* `y` is local to the nested function, and unknown to the outer function. The example on the right assigns the return value to a variable, `z`, and then displays the value of `z` correctly.

Incorrect	Correct

`function varScope4`
`x = 5; nestfun;`
`   function y = nestfun`
`      y = x + 1;`
`   end`
 	
`y`
`end`
	`function varScope5`
	`x = 5;`
	`z = nestfun;`
	`   function y = nestfun`
	`      y = x + 1;`
	`   end`
	
	`z`
	`end`

Using Function Handles with Nested Functions

Every function has a certain *scope*, that is, a certain range of other functions to which it is visible. A function's scope determines which other functions can call it. You can call a function that is out of scope by providing an alternative means of access to it in the form of a function handle. (The function handle, however, must be within the scope of its related function when you construct the handle.) Any function that has access to a function handle can call the function with which the handle is associated.

Note Although you can call an out of scope function by means of a function handle, the handle itself must be within the scope of its related function at the time it is constructed.

The section on "Calling Nested Functions" on page 4-18 defines the scope of a nested function. As with other types of functions, you can make a nested function visible beyond its normal scope with a function handle. The following function `getCubeHandle` constructs a handle for nested function `findCube`

and returns its handle, h, to the caller. The @ sign placed before a function name (e.g., @findCube) is the MATLAB operator that constructs a handle for that function:

```
function h = getCubeHandle
h = @findCube;                     % Function handle constructor

   function cube = findCube(X)     % Nested function
      cube = X .^ 3;
   end
end
```

Call getCubeHandle to obtain the function handle to the nested function findCube. Assign the function handle value returned by getCubeHandle to an output variable, cubeIt in this case:

```
cubeIt = getCubeHandle;
```

You can now use this variable as a means of calling findCube from outside of its M-file:

```
cubeIt(8)
ans =
    512
```

Note When calling a function by means of its handle, use the same syntax as if you were calling a function directly. But instead of calling the function by its name (e.g., strcmp(S1, S2)), use the variable that holds the function handle (e.g., fhandle(S1, S2)).

Function Handles and Nested Function Variables

One characteristic of nested functions that makes them different from other MATLAB functions is that they can share nonglobal variables with certain other functions within the same M-file. A nested function nFun can share variables with any outer function that contains nFun, and with any function nested within nFun. This characteristic has an impact on how certain variables are stored when you construct a handle for a nested function.

Defining Variables When Calling Via Function Handle. The example below shows a primary function `getHandle` that returns a function handle for the nested function `nestFun`. The `nestFun` function uses three different types of variables. The `VLoc` variable is local to the nested function, `VInp` is passed in when the nested function is called, and `VExt` is defined by the outer function:

```
function h = getHandle(X)
h = @nestFun;
VExt = someFun(X);

   function nestFun(VInp)
   VLoc = 173.5;
   doSomeTask(VInp, VLoc, VExt);
   end
end
```

As with any function, when you call `nestFun`, you must ensure that you supply the values for any variables it uses. This is a straightforward matter when calling the nested function directly (that is, calling it from `getHandle`). `VLoc` has a value assigned to it within `nestFun`, `VInp` has its value passed in, and `VExt` acquires its value from the workspace it shares with `getHandle`.

However, when you call `nestFun` using a function handle, only the nested function executes; the outer function, `getHandle`, does not. It might seem at first that the variable `VExt`, otherwise given a value by `getHandle`, has no value assigned to it in the case. What in fact happens though is that MATLAB stores variables such as `VExt` inside the function handle itself when it is being constructed. These variables are available for as long as the handle exists.

The `VExt` variable in this example is considered to be *externally scoped* with respect to the nested function. Externally scoped variables that are used in nested functions for which a function handle exists are stored within the function handle. So, function handles not only contain information about accessing a function. For nested functions, a function handle also stores the values of any externally scoped variables required to execute the function.

Example Using Externally Scoped Variables

The `sCountFun` and `nCountFun` functions shown below return function handles for subfunction `subCount` and nested function `nestCount`, respectively.

These two inner functions store a persistent value in memory (the value is retained in memory between function calls), and then increment this value on every subsequent call. `subCount` makes its `count` value persistent with an explicit `persistent` declaration. In `nestCount`, the count variable is externally scoped and thus is maintained in the function handle:

Using a Subfunction	Using a Nested Function

`function h = sCountFun(X)`
`h = @subCount;`
`count = X`
`subCount(0, count);`
`function subCount(incr, ini)`
`persistent count;`
`initializing = nargin > 1;`
`if initializing`
`   count = ini; else`
`   count = count + incr`
`end`
	`function h = nCountFun(X)`
	`h = @nestCount;`
	`count = X`
	`   function nestCount(incr)`
	`      count = count + incr`
	`   end`
	`end`

When `sCountFun` executes, it passes the initial value for `count` to the `subCount` subfunction. Keep in mind that the `count` variable in `sCountFun` is not the same as the count variable in `subCount`; they are entirely independent of each other. Whenever `subCount` is called via its function handle, the value for `count` comes from its persistent place in memory.

In `nestCount`, the `count` variable again gets its value from the primary function when called from within the M-file. However, in this case the `count` variable in the primary and nested functions are one and the same. When `nestCount` is called by means of its function handle, the value for `count` is assigned from its storage within the function handle.

Running the Example. The `subCount` and `nestCount` functions increment a value in memory by another value that you pass as an input argument. Both of these functions give the same results.

Get the function handle to `nestCount`, and initialize the `count` value to a four-element vector:

```
h = nCountFun([100 200 300 400])
```

```
count =
   100   200   300   400
```

Increment the persistent vector by 25, and then by 42:

```
h(25)
count =
   125   225   325   425

h(42)
count =
   167   267   367   467
```

Now do the same using `sCountFun` and `subCount`, and verify that the results are the same.

Note If you construct a new function handle to `subCount` or `nestCount`, the former value for `count` is no longer retained in memory. It is replaced by the new value.

Separate Instances of Externally Scoped Variables

The code shown below constructs two separate function handles to the same nested function, `nestCount`, that was used in the last example. It assigns the handles to fields `counter1` and `counter2` of structure `s`. These handles reference different instances of the `nestCount` function. Each handle also maintains its own separate value for the externally scoped `count` variable.

Call `nCountFun` twice to get two separate function handles to `nestCount`. Initialize the two instances of `count` to two different vectors:

```
s.counter1 = nCountFun([100 200 300 400]);
count =
   100   200   300   400

s.counter2 = nCountFun([-100 -200 -300 -400]);
count =
  -100  -200  -300  -400
```

Now call `nestCount` by means of each function handle to demonstrate that MATLAB increments the two `count` variables individually.

Increment the first counter:

```
s.counter1(25)
count =
   125   225   325   425
s.counter1(25)
count =
   150   250   350   450
```

Now increment the second counter:

```
s.counter2(25)
count =
   -75  -175  -275  -375
s.counter2(25)
count =
   -50  -150  -250  -350
```

Go back to the first counter and you can see that it keeps its own value for `count`:

```
s.counter1(25)
count =
   175   275   375   475
```

Restrictions on Assigning to Variables

The scoping rules for nested, and in some cases anonymous, functions require that all variables used within the function be present in the text of the M-file code. Adding variables to the workspace of this type of function at run time is not allowed.

MATLAB issues an error if you attempt to dynamically add a variable to the workspace of an anonymous function, a nested function, or a function that contains a nested function. Examples of operations that might use dynamic assignment in this way are shown in the table below.

Type of Operation	How to Avoid Using Dynamic Assignment
Evaluating an expression using `eval` or `evalin`, or assigning a variable with `assignin`	As a general suggestion, it is best to avoid using the `eval`, `evalin`, and `assignin` functions altogether.
Loading variables from a MAT-file with the `load` function	Use the form of `load` that returns a MATLAB structure.
Assigning to a variable in a MATLAB script	Convert the script to a function, where argument- and result-passing can often clarify the code as well.
Assigning to a variable in the MATLAB debugger	You can declare the variable to be `global`. For example, to create a variable X for temporary use in debugging, use `K>> global X;    X = value`

One way to avoid this error in the other cases is to pre-declare the variable in the desired function.

Examples of Nested Functions

This section shows a few examples of how you can use nested functions. These examples are intended to show you how to program with this type of function. For more mathematically oriented examples, see the MATLAB Mathematics documentation.

The examples in this section include

- "Example 1 — Creating a Function Handle for a Nested Function" on page 4-27
- "Example 2 — Function-Generating Functions" on page 4-29

Example 1 — Creating a Function Handle for a Nested Function

The following example constructs a function handle for a nested function and then passes the handle to the MATLAB `fplot` function to plot the parabola

shape. The makeParabola function shown here constructs and returns a function handle fhandle for the nested parabola function. This handle gets passed to fplot:

```
function fhandle = makeParabola(a, b, c)
% MAKEPARABOLA returns a function handle with parabola
% coefficients.

fhandle = @parabola;     % @ is the function handle constructor

   function y = parabola(x)
   y = a*x.^2 + b*x + c;
   end
end
```

Assign the function handle returned from the call to a variable (h) and evaluate the function at points 0 and 25:

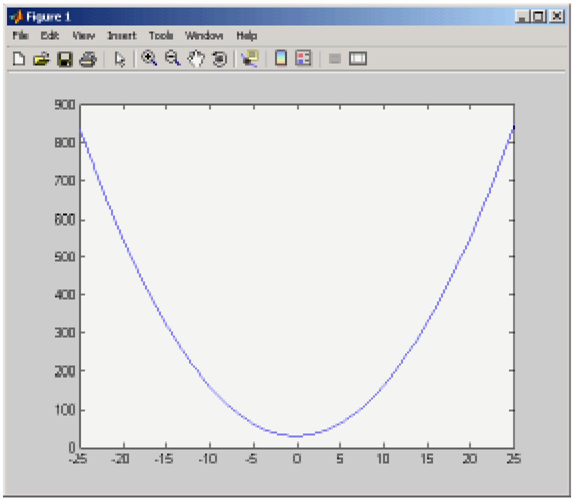
```
h = makeParabola(1.3, .2, 30)
h =
    @makeParabola/parabola

h(0)
ans =
    30

h(25)
ans =
  847.5000
```

Now pass the function handle h to the fplot function, evaluating the parabolic equation from x = -25 to x = +25:

```
fplot(h, [-25 25])
```



Example 2 — Function-Generating Functions

The fact that a function handle separately maintains a unique instance of the function from which it is constructed means that you can generate multiple handles for a function, each operating independently from the others. The function in this example makes IIR filtering functions by constructing function handles from nested functions. Each of these handles maintains its own internal state independent of the others.

The function makeFilter takes IIR filter coefficient vectors a and b and returns a filtering function in the form of a function handle. Each time a new input value x_n is available, you can call the filtering function to get the new output value y_n. Each filtering function created by makeFilter keeps its own private a and b vectors, in addition to its own private state vector, in the form of a transposed direct form II delay line:

```
function [filtfcn, statefcn] = makeFilter(b, a)
%   FILTFCN = MAKEFILTER(B, A) creates an IIR filtering
%   function and returns it in the form of a function handle,
```

```
%   FILTFCN. Each time you call FILTFCN with a new filter
%   input value, it computes the corresponding new filter
%   output value, updating its internal state vector at the
%   same time.
%
%   [FILTFCN, STATEFCN] = MAKEFILTER(B, A) also returns a
%   function (in the form of a function handle, STATEFCN)
%   that can return the filter's internal state.  The internal
%   state vector is in the form of a transposed direct form
%   II delay line.

%   Initialize state vector. To keep this example a bit
%   simpler, assume that a and b have the same length.
%   Also assume that a(1) is 1.

v = zeros(size(a));

filtfcn =  @iirFilter;
statefcn = @getState;

   function yn = iirFilter(xn)
      % Update the state vector
      v(1) = v(2) + b(1) * xn;
      v(2:end-1) = v(3:end) + b(2:end-1) * xn - ...
         a(2:end-1) * v(1);
      v(end) = b(end) * xn - a(end) * v(1);

      % Output is the first element of the state vector.
      yn = v(1);
   end

   function vOut = getState
      vOut = v;
   end
end
```

This sample session shows how `makeFilter` works. Make a filter that has a decaying exponential impulse response and then call it a few times in succession to see the output values change:

```
[filt1, state1] = makeFilter([1 0], [1 -.5]);

% First input to the filter is 1.
filt1(1)
ans =
   1

% Second input to the filter is 0.
filt1(0)
ans =
   0.5000

filt1(0)
ans =
   0.2500

% Show the filter's internal state.
state1()
ans =
   0.2500    0.1250

% Hit the filter with another impulse.
filt1(1)
ans =
   1.1250

% How did the state change?
state1()
ans =
   1.1250    0.5625

% Make an averaging filter.
filt2 = makeFilter([1 1 1]/3, [1 0 0]);

% Put a step input into filt2.
filt2(1)
ans =
   0.3333

filt2(1)
```

```
ans =
   0.6667

filt2(1)
ans =
   1

% The two filter functions can be used independently.
filt1(0)
ans =
   0.5625
```

As an extension of this example, suppose you were looking for a way to develop simulations of different filtering structures and compare them. This might be useful if you were interested in obtaining the range of values taken on by elements of the state vector, and how those values compare with a different filter structure. Here is one way you could capture the filter state at each step and save it for later analysis:

Call `makeFilter` with inputs `v1` and `v2` to construct function handles to the `iirFilter` and `getState` subfunctions:

```
[filtfcn, statefcn] = makeFilter(v1, v2);
```

Call the `iirFilter` and `getState` functions by means of their handles, passing in random values:

```
x = rand(1, 20);
for k = 1:20
    y(k) = filtfcn(x(k));
    states{k} = statefcn();  % Save the state at each step.
end
```

Subfunctions

In this section...

Overview

M-files can contain code for more than one function. Additional functions within the file are called *subfunctions*, and these are only visible to the primary function or to other subfunctions in the same file.

Each subfunction begins with its own function definition line. The functions immediately follow each other. The various subfunctions can occur in any order, as long as the primary function appears first:

```
function [avg, med] = newstats(u) % Primary function
% NEWSTATS Find mean and median with internal functions.
n = length(u);
avg = mean(u, n);
med = median(u, n);

function a = mean(v, n)            % Subfunction
% Calculate average.
a = sum(v)/n;

function m = median(v, n)          % Subfunction
% Calculate median.
w = sort(v);
if rem(n, 2) == 1
   m = w((n+1) / 2);
else
   m = (w(n/2) + w(n/2+1)) / 2;
end
```

The subfunctions `mean` and `median` calculate the average and median of the input list. The primary function `newstats` determines the length of the list and calls the subfunctions, passing to them the list length `n`.

Subfunctions cannot access variables used by other subfunctions, even within the same M-file, or variables used by the primary function of that M-file, unless you declare them as global within the pertinent functions, or pass them as arguments.

Calling Subfunctions

When you call a function from within an M-file, MATLAB first checks the file to see if the function is a subfunction. It then checks for a private function (described in the following section) with that name, and then for a standard M-file or built-in function on your search path. Because it checks for a subfunction first, you can override existing M-files using subfunctions with the same name.

Accessing Help for a Subfunction

You can write help for subfunctions using the same rules that apply to primary functions. To display the help for a subfunction, precede the subfunction name with the name of the M-file that contains the subfunction (minus file extension) and a > character.

For example, to get help on subfunction `mysubfun` in file `myfun.m`, type

```
help myfun>mysubfun
```

Private Functions

In this section...

Overview

Private functions are functions that reside in subdirectories with the special name `private.` These functions are called *private* because they are visible only to M-file functions and M-file scripts that meet these conditions:

- A function that calls a private function must be defined in an M-file that resides in the directory immediately above that `private` subdirectory.
- A script that calls a private function must itself be called from an M-file function that has access to the private function according to the above rule.

For example, assume the directory `newmath` is on the MATLAB search path. A subdirectory of `newmath` called `private` can contain functions that only the functions in `newmath` can call.

Because private functions are invisible outside the parent directory, they can use the same names as functions in other directories. This is useful if you want to create your own version of a particular function while retaining the original in another directory. Because MATLAB looks for private functions before standard M-file functions, it will find a private function named `test.m` before a nonprivate M-file named `test.m`.

Primary functions and subfunctions can also be implemented as private functions.

Private Directories

You can create your own private directories simply by creating subdirectories called `private` using the standard procedures for creating directories or folders on your computer. Do not place these private directories on your path.

Accessing Help for a Private Function

You can write help for private functions using the same rules that apply to primary functions. To display the help for a private function, precede the private function name with `private/`.

For example, to get help on private function `myprivfun`, type

```
help private/myprivfun
```

Overloaded Functions

Overloaded functions are useful when you need to create a function that responds to different types of inputs accordingly. For instance, you might want one of your functions to accept both double-precision and integer input, but to handle each type somewhat differently. You can make this difference invisible to the user by creating two separate functions having the same name, and designating one to handle `double` types and one to handle integers.

MATLAB overloaded functions reside in subdirectories having a name starting with the symbol `@` and followed by the name of a recognized MATLAB class. For example, functions in the `\@double` directory execute when invoked with arguments of MATLAB type `double`. Those in an `\@int32` directory execute when invoked with arguments of MATLAB type `int32`.

See "Overloading MATLAB Functions" for more information on overloading functions in MATLAB.

Using Objects

MATLAB Objects

In this section...

What Are Objects?

This chapter provides information for people using objects. It does not provide a thorough treatment of object-oriented concepts, but instead focuses on what you need to know to use the objects provided with MATLAB.

If you are interested in object-oriented programming in the MATLAB language, see *Object-Oriented Programming*. For background information on objects, see object-oriented design.

In the simplest sense, objects are special-purpose variables. Objects differ from general-purpose variables in that they define the operations you can perform on the data they contain. These operations create an interface with which you interact with the object, without needing to know how operations are implemented or how data is stored. This makes objects modular and easy to pass within application programs. It also isolates your code from changes to the object's design and implementation.

In a more general sense, objects are organized collections of data and functions that have been designed for specific purposes. For example, an object might be designed to contain time series data that consists of value/time-sample pairs and associated information like units, sample uniformity, and so on. This object could have a set of specific operations designed to perform analysis that is relevant to this particular type of data. The following sections provide examples of such objects.

Accessing Objects

You access an object with its variable name. Interacting with objects variables in MATLAB software is really no different from interacting with any other variables. Basically, you can perform the same common operations on

variables whether they hold numbers or specialized objects. For example, you can do the following things with objects:

- Create it and assigned a variable name so you can reference it again
- Assign or reassign data to it (see "Accessing Object Data" on page 5-14)
- Operate on its data (see "Calling Object Methods" on page 5-16)
- Convert it to another class (if this operation is supported by the object's class)
- Save it to a MAT-file so you can reload it later (see `save`)
- Copy it (see "Copying Objects" on page 5-26)
- Clear it from the workspace (`clear`)

Any particular object might have restrictions on how you create it, access its data, or what operations you can perform on it. Refer to the documentation for the particular MATLAB object for a description of what you can do with that object.

See "Variables" on page 2-9 for a general discussion of MATLAB variables.

Objects In the MATLAB Language

The MATLAB language uses many specialized objects. For example, `timer` objects execute code at a certain time interval, `MException` objects capture information when errors occur, the `serial` object enables you to communicate with devices connected to your computer's serial port, and so on. MATLAB toolboxes often define objects to manage the specific data and analyses performed by the toolbox.

All of these objects are designed to provide specific functionality that is not as conveniently available from general purpose language components.

Other Kinds of Objects Used by MATLAB

The MATLAB language enables you to use other kinds of objects in your MATLAB programs. The following objects are different from the MATLAB objects described in this documentation. See the individual sections referenced below for information on using these objects.

- Handle Graphics® objects represent objects used to create graphs and GUIs. These objects provide a `set/get` interface to property values, but are not extensible by subclassing. See "Handle Graphics Objects" for more information.
- Sun Java objects can be used in MATLAB code enabling you to access the capabilities of Java classes. See "Using Sun Java Classes in MATLAB Software" for more information.
- Microsoft COM objects enable you to integrate these software components into your application. See "COM Support for MATLAB Software" for more information.
- User-defined MATLAB objects created prior to Version 7.6 used different syntax for class definition (no `classdef` block) and exhibit other differences. See "Compatibility with Previous Versions " for more information.

General Purpose Vs. Specialized Arrays

In this section...

How They Differ

The MATLAB language enables you to use both general-purpose and specialized arrays. For example, numeric multidimensional arrays and structures provide general-purpose data storage. You typically extract data from the array and perform operations (e.g., mathematical analysis) on this data, and then store the data back in general-purpose variables.

When using a specialized object, you typically pass the object's data to a function that creates the object. Once you have created the object, you use specially defined functions to operate on the data. These functions are unique to the object and are designed specifically for the type and structure of the data contained by the object.

Using General-Purpose Variables

A commonly used general-purpose variable is a structure array. For example, these statements create a MATLAB `struct` (a MATLAB structure array):

```
s.Data = rand(10,1);
s.Time = .01:.01:.1;
s.Name = 'Data1';
s.Units = 'seconds;
```

The structure `s` contains two arrays of numbers. However, `s` is a generic variable in the sense that MATLAB does not define special functions to operate on the data in this particular structure. For example, while `s` contains two fields, `Data` and `Time`, that would be useful to plot, you cannot pass `s` to the `plot` function:

```
plot(s)
```

```
??? Error using ==> plot
Not enough input arguments.
```

While `s` certainly has enough information to create a plot of `Data` versus `Time`, `plot` cannot access this data because structures like `s` can contain any values in its fields and the fields can have any name. Just because one field is named `Data` does not force you to assign data to that field.

To plot the data in `s`, you would have to extract the data from the fields, pass them as arguments in the desired order to the `plot` function, add a title, labels, and so on:

```
plot(s.Time,s.Data)
title(['Time Series Plot: ' s.Name])
xlabel(['Time (' s.Units ')'])
ylabel(s.Name)
```

You could create a function to perform these steps for you. Other programs using the structure `s` would need to create their own functions or access the one you created.

Using Specialized Objects

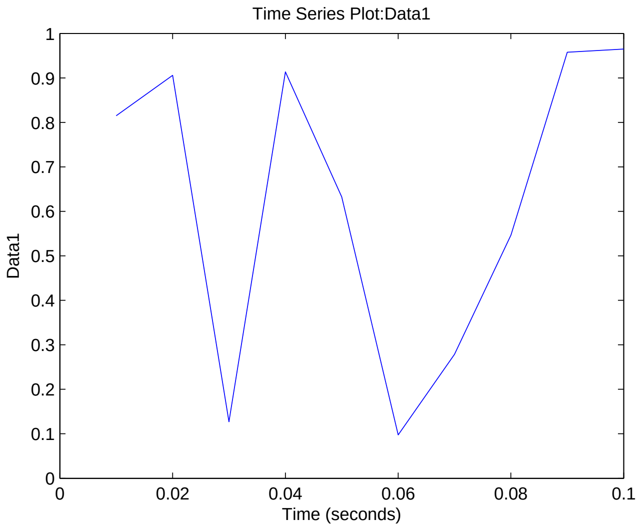
Compare the structure array above to an object that has been specifically designed to contain and manipulate time series data. For example, the following statement creates a MATLAB `timeseries` object. It is initialized to store the same data as structure `s` above:

```
tsobj = timeseries(rand(10,1),.01:.01:.1,'Name','Data1');
```

The function that creates the object `tsobj`, accepts sample data, sample times, a property name/property value pair (`Name/Data1`), and uses a default value of `Units` (which is `seconds`).

The designer of this object created a special version of the `plot` function that works directly with this object. For example:

```
plot(tsobj)
```



Notice how the object's `plot` function creates a graph that is plotted and labeled with the data from the `tsobj` object. As a user of this object, you do not need write your own code to produce this graph. The class design specifies the standard way to present graphs of `timeseries` data and all clients of this object use the same code for plotting.

See "Time Series Objects" for more on using MATLAB `timeseries` objects.

Key Object Concepts

In this section...
"Basic Concepts" on page 5-8
"Classes Describe How to Create Objects" on page 5-8
"Properties Contain Data" on page 5-8
"Methods Implement Operations" on page 5-9

Basic Concepts

There are some basic concepts that are fundamental to objects. Objects represent something in the real world, like an error condition or the set of data you collected in a product test. Objects enable you to do something useful, like provide an error report or analyze and present the results of tests.

This section introduces the basic components that MATLAB uses to realize the design of an object. These components include:

- Classes
- Properties
- Methods

Classes Describe How to Create Objects

A *class* defines a set of similar objects. It is a description from which MATLAB creates a particular instance of the class, and it is the instance (that is, the object) that contains actual data. Therefore, while there is a `timeseries` class, you work with `timeseries` objects.

Classes are defined in code files — either as separate M-files or built-in to the MATLAB executable. Objects are specific representations of a class that you access through workspace variables.

Properties Contain Data

Objects store data in *properties*. Consider a `timeseries` object as an example. A `timeseries` object stores data values, associated time values, and also

related information, such as units, events, data quality, and interpolation method. All this data is stored in various object properties. MATLAB objects enable you to access property data directly (see "Accessing Object Data" on page 5-14 for information on property syntax).

Properties are sometimes called fields in other programming languages and are similar to the fields of MATLAB structures. Properties have descriptive names, such as `Data` and `DataInfo`, in the case of `timeseries` objects, and can contain any kind of MATLAB data, including other objects.

An object, then, is a container for a predefined set of data. Unlike a cell array or structure, you cannot add new properties or delete defined properties from an object. Doing so would compromise the object's intended purpose and violate the class design.

The class design can restrict the values you can assign to a property. For example, a `Length` property might restrict possible values to positive integers or might be read only and determine its own value when queried.

Methods Implement Operations

Class *methods* are functions designed to work with objects of a particular class. Methods enable the class designer to implement specific operations that are optimized for the data contained in the object. You do not have to extract the data from the object, modify its format, and pass it to a general-purpose MATLAB function because the class defines methods with an awareness of the object's structure.

Methods can define operations that are unique to a particular class of object, such as adding a data sample to an existing set of time series data, or can *overload* common operations in a way that makes sense for the particular object. For example, `timeseries` objects have an `addsample` method to add a new data sample to an existing `timeseries` object. Also, `timeseries` overloads the MATLAB `plot` function to work with `timeseries` objects.

MATLAB software determines which overloaded version of a method to call based on the class of the object passed as an argument. If you execute a MATLAB statement like:

```
tsobjnew = tsobj1 + tsobj2;
```

where tsobj1 and tsobj2 are timeseries objects, MATLAB calls the timeseries version of the + operation (if defined) and returns a new timeseries object.

Because the timeseries class defines the operation, you can add a timeseries object to a scalar number:

```
tsobjnew = tsobj1 + 4;
```

The class definition determines what happens when you add a scalar double to a timeseries object (the scalar is added to each Data value).

Methods make working with objects convenient for the user, but also provide advantages to the class designer. Methods hide implementation details from users—you do not need to create your own functions to access and manipulate data, as you would when using general-purpose data structures like structs and cell arrays. This provides the flexibility to change the internal design of an object without affecting object clients (i.e., application programs that use the objects).

Creating Objects

In this section...

Class Constructor

Usually, you create an object by calling a function designed for the purpose of creating that specific class of object. For example, the following code creates a `timeseries` object and assigns it to the variable `tsboj`:

```
load count.dat % Load some data
tsobj = timeseries(count(:,1),1:24,'Name','Data1');
```

The `timeseries` method creates an object and initializes its data with the values specified as arguments. Classes that create objects define a special method whose purpose is to create objects of the class. This function has the same name as the class and is called the *class constructor*.

However, in some cases, you might create objects by calling other functions or even using a GUI. For example, a `try-catch` block can return an `MException` object that contains information about a specific error condition. In this case, you do not explicitly create the object, rather it is returned by the `catch` statement (see "Accessing Object Data" on page 5-14 for an example).

When to Use Package Names

A package is a container that provides a logical grouping for class and function definitions. The class and function names within a given package must be unique, but can be reused in other packages. Packages are directories that begin with the + character.

If a package directory contains a class definition, then you must use the package name when calling the class constructor. For example, this statement creates a `Map` object, whose class definition file is in a directory in the `containers` package:

```
mapobj = containers.Map({'rose','bicycle'},{'flower','machine'});
```

You need to use the package name to refer to:

- Class constructors (e.g., `containers.Map`), which you call to create an object
- Static methods (methods that do not require an object of the class as an argument)
- Package functions (functions defined in the package)

However, because MATLAB uses the class of an object to determine which ordinary method to call, you do not need to use the package name in conjunction with object references. For example, suppose you have the following directory structure:

```
pathdirectory/+packagename/@ClassName/ClassName.m
pathdirectory/+packagename/@ClassName/staticMethodName.m
pathdirectory/+packagename/functionName.m
```

In the following examples, `obj` is the object you are creating.

```
% Create object of ClassName
obj = packagename.ClassName(...);

% Call methodName
obj.methodName(...);

% Set or get the value of property PropertyName
obj.PropertyName = x;
x = obj.PropertyName;

% Call static method staticMethodName
packagename.ClassName.staticMethodName(...);

% Call package function functionName
packagename.functionName(...)
```

If a package directory contains a class definition file, then consider the package name as part of the class name. Wherever you need to use the class name, include the package name. For example, `containers.Map` is the full class name of the `Map` class.

See the object's user documentation for the syntax you need to use to create objects.

See "Organizing Classes in Directories" and "Scoping Classes with Packages" for more information on the use of packages.

See "Importing Classes" for information on importing packages into functions.

Accessing Object Data

In this section...

Listing Public Properties

Note You should always treat property names as being case sensitive.

You can see the names of all public object properties using the `properties` function with the object's class name or with an actual object. For example:

```
>> properties('MException')
Properties for class MException:
    identifier
    message
    cause
    stack
```

Getting Property Values

After creating an object, you can access the values of its properties:

```
try
   a = rand(4);
   a(17) = 7;
catch me % catch creates an MException object named me
   disp(['Current error identifier: ' me.identifier])
end
Current error identifier: MATLAB:indexed_matrix_cannot_be_resized
```

Access the data in properties using dot notation:

```
object.PropertyName
```

For example, you can access the message property of the MException object, me, with this syntax:

```
>> me.message
ans =
 In an assignment  A(I) = B, a matrix A cannot be resized.
```

See "Capturing Information About the Error" on page 8-5 for more information on using MException objects.

Setting Property Values

Objects often restrict what values you can assign to them. For example, the following timeseries object has 10 data values, each corresponding to a sample time:

```
tsobj = timeseries(rand(10,1),1:10,'Name','Random Sample');
```

Now suppose you attempt to set the Data property to a three-element vector:

```
>> tsobj.Data = [1 2 3];
??? Error using ==> timeseries.timeseries>timeseries.set.Data at 171
Size of the data array is incompatible with the time vector.
```

The timeseries class design ensures that the number of data samples matches the number of time samples. This illustrates one of the advantages a specialized object has over a general purpose-data structure like a MATLAB struct.

Calling Object Methods

In this section...

What Operations Can You Perform

Methods define all aspects of an object's behavior. Consequently, most classes implement many methods that an object user is unlikely to call directly. The user documentation for the object you are using describes the operations you can perform on any particular object.

You can list the methods defined by a class with the `methods` or `methodsview` functions:

```
methods('timeseries')

Methods for class timeseries:
addevent              gettsbetweenevents    set
addsample             horzcat               setabstime
createTstoolNode      idealfilter           setinterpmethod
ctranspose            init                  setprop
...
gettsatevent          pvset                 var
gettsbeforeatevent    rdivide               vertcat
gettsbeforeevent      resample

Static methods:
tsChkTime             tsgetrelativetime
```

Method Syntax

You call an object's method using dot notation:

```
returnedValue = object.MethodName(args,...)
```

You also can call a method using function syntax, passing the object as the first (left-most) argument.

For example, `MException` objects have a `getReport` method that returns information about the error.

```
try
   surf
catch me
   disp(me.getReport)
end

Error using ==> surf at 50
Not enough input arguments.
```

Dot and function notation are usually equivalent. That is, both of the following statements return the `MException` report:

```
rpt = getReport(me); % Call getReport using function notation
rpt = me.getReport;  % Call getReport using dot notation
```

Calling the Correct Method

It is possible for the function syntax to call an unexpected method if there is more than one object in the argument list. Suppose there are two classes, `ClassA` and `ClassB`, that define a method called `addData`. Suppose further that `ClassA` is defined as being inferior to `ClassB` in precedence (something that the class designer can do in the class definition). In this situation, given `objA` is of `ClassA` and `objB` is of `ClassB`, the following two statement call different methods:

```
addData(objA,objB) % Calls objB.addData
objA.addData(objB) % Calls objA.addData
```

If `ClassA` and `ClassB` had equal precedence, then the left-most argument determines which method MATLAB calls (i.e., `objA.addData` in both statements).

It is unlikely that you will encounter this particular scenario, however, if you are calling a method that accepts more than one object as arguments, using

dot notation removes any ambiguity about which object's method MATLAB calls.

Class of Objects Returned by Methods

While methods sometimes return objects of the same class, this is not always the case. For example, the `MException` object's `getReport` returns a character string:

```
try
   surf
catch me
   rpt = me.getReport;
end

whos
  Name        Size              Bytes  Class         Attributes
  me          1x1                 780  MException
  rpt         1x171               342  char
```

Methods can return any type of value and properties can contain any type of value. However, class constructor methods always return an object or array of objects of the same type as the class.

Desktop Tools Are Object Aware

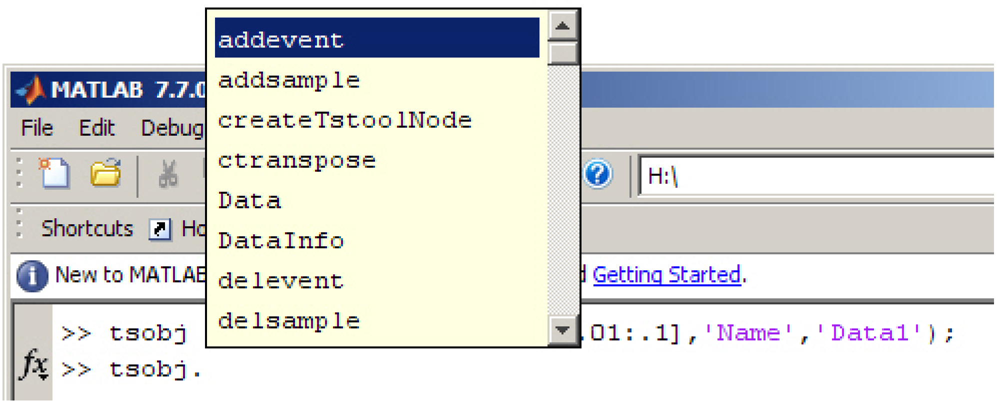
In this section...

Tab Completion Works with Objects

MATLAB tab completion works with objects. For example, if you enter an object name followed by a dot:

```
tsobj.
```

and then press the tab key, MATLAB pops up a selection box with a list of properties and methods:



The more letters you complete after the dot, the more specific is the list. See "Completing Statements in the Command Window — Tab Completion" for more information.

Editing Objects with the Variable Editor

You can use the MATLAB Variable Editor to edit object properties. To open an object in the Variable Editor, you can double-click the object name in the Workspace browser or use the `openvar` command:

```
tsobj = timeseries(rand(10,1),.01:.01:.1,'Name','Data1');
openvar tsobj
```

See “Viewing and Editing Workspace Variables with the Variable Editor” for more information.

Getting Information About Objects

In this section...

The Class of Workspace Variables

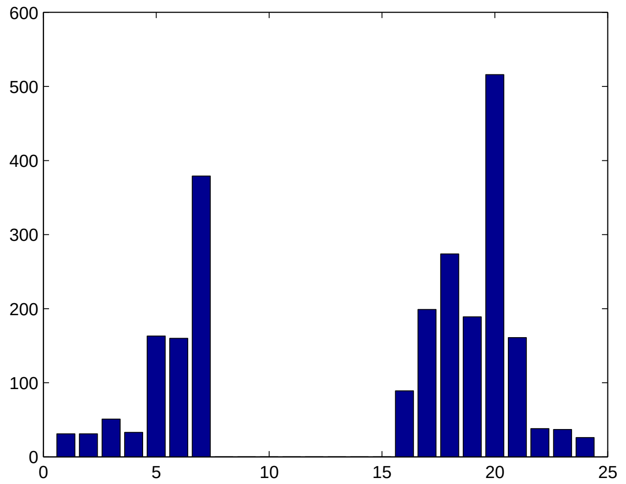
Knowing the class of the variables you are working with enables you to use them most effectively. For example, consider the following variable created in your workspace:

```
load count.dat % Load some data
tsobj = timeseries(count(:,1),1:24,'Name','Data1');
>> whos
  Name        Size            Bytes  Class
  count      24x3               576  double
  tsobj      24x1               261  timeseries
```

The `whos` command lists information about your workspace variables. Notice that the variable loaded from the `count.dat` file (`count`) is an array of doubles. You know, therefore, that you can perform indexing and arithmetic operations on this array. For example:

```
newcount = sum(count,2);
newcount(8:15) = NaN;
bar(newcount)
```

Indexed assignment and the `bar` function work with inputs of class `double`.



However, the `timeseries` class does not define a `bar` method for `timeseries` objects. The `timeseries` class defines a `plot` method for graphing because the class design specified a line plot as the best way to represent time series data.

Extracting Data From Object Properties

Suppose you have a `timeseries` object and you want to work directly with the numeric values of the `timeseries` data. You can extract data from the object properties and assign these values to an array. For example

```
load count
tsobj = timeseries(sum(count,2),1:24,'Name','DataSum');
d = tsobj.Data;
t = tsobj.Time;
n = tsojb.Name;
d(8:15) = NaN;
bar(t,d); title(n)
```

Testing for the Class of an Object

Suppose you create an M-file that operates on more than one class of object. If you have a `timeseries` object, you call the `timeseries` `plot` method, but

if the object is of class `double`, you can call the `bar` function (which isn't supported by `timeseries` objects). You could use `isa` as in the following code to make this determination:

```
if isa(obj,'timeseries')
   plot(obj)
elseif isa(obj,'double')
   bar(obj)
end
```

Information About Class Members

These functions provide information about the object.

Function	Purpose
`class`	Return class of object
`events`	List of event names defined by the class
`methods`	List of methods implemented by the class
`methodsview`	Information on class methods in separate window
`properties`	List of class property names

Logical Tests for Objects

In M-files, you might need conditional statements to determine the status of an object before performing certain actions. For example, you might perform different actions based on the class of an object (see "Testing for the Class of an Object" on page 5-22). The following functions provide logical tests for objects:

Function	Purpose
`isa`	Determine whether argument belongs to a particular class. True for object's class and all of object's superclasses.
`isequal`	Determine if two objects are equal.
`isobject`	Determine whether the input is a MATLAB object.

Testing for Object Equality

`isequal` finds two objects to be equal if all the following conditions are met:

- Both objects are of the same class
- Both objects are of the same size
- All corresponding property values are equal

`isequal` tests the value of every array element in every property and every property of every object contained in the objects being tested. As contained objects are tested for equality, MATLAB calls each object's own version of `isequal` (if such versions exist).

If objects contain large amounts of data stored in other objects, then testing for equality can be a time-consuming process.

Identifying MATLAB Objects

The `isobject` function returns `true` only for MATLAB objects. For Sun Java objects, use `isjava`. For Handle Graphics objects, use `ishandle`.

Note `ishandle` returns false for MATLAB handle objects. See "Testing for Handle or Value Class" on page 5-30 for more information.

Displaying Objects

When you issue commands that return objects and do not terminate those commands with a semicolon, or when you pass an object to the `disp` function, MATLAB displays information about the object. For example:

```
hobj = containers.Map({'Red Sox','Yankees'},
{'Boston','New York'})
hobj =
  containers.Map handle
  Package: containers
  Properties:
        Count: 2
      KeyType: 'char'
    ValueType: 'char'
```

```
    Methods, Events, Superclasses
```

This information includes links (shown in blue) to documentation on the object's class and superclasses, and lists of methods, events, and superclasses. Properties and their current values are also listed.

Some classes (`timeseries`, for example) redefine how they display objects to provide more useful information for this particular class.

Getting Help for MATLAB Objects

You can get documentation for MATLAB objects using the `doc` command with the class name. To see the reference pages for the objects used in this chapter, use the following commands:

```
doc timeseries
doc MException
doc containers.Map % Include the package name
```

Copying Objects

In this section...

Two Copy Behaviors

There are two fundamental kinds of MATLAB classes—handles and values.

Value classes create objects that behave like ordinary MATLAB variables with respect to copy operations. Copies are independent values. Operations that you perform on one object do not affect copies of that object.

Handle classes create objects that are sometimes referred to as references. This is because a handle, and all copies of this handle, refer to the same underlying object. When you create a handle object, you can copy the handle, but not the data referenced by the object's properties. Any operations you perform on a handle object affects all copies of that object. Handle Graphics objects behave in this way.

More Information About Handle and Value Classes

For more detailed information about handle and value classes, see "Value or Handle Class — Which to Use" in the Object-Oriented Programming documentation.

Value Object Copy Behavior

MATLAB numeric variables exhibit the behavior of value objects. For example, when you copy `a` to the variable `b`, both variables are independent of each other. Changing the value of `a` does not change the value of `b`:

```
a = 8;
b = a;
```

Now reassign `a` and `b` is unchanged:

```
a = 6;
b
b =
     8
```

Clearing `a` does not affect `b`:

```
clear a
b
b =
     8
```

Value Object Properties

The copy behavior of values stored as properties in value objects is the same. For example, suppose `vobj1` is a value object with property `a`:

```
vobj1.a = 8; % Property is set to a value
```

If you copy `vobj1` to `vobj2`, and then change the value of `vobj1` property `a`, you can see that the value of the copied object's property `vobj2.a` is unaffected:

```
vobj2 =vobj1;
vobj1.a = 5;

vobj2.a
ans =
     8
```

Handle Object Copy Behavior

Suppose you have a handle class called `HdClass` that defines a property called `Data`, and that you create an object of this class with the following statement:

```
hobj1 = HdClass(8)
```

Because this statement is not terminated with a semicolon, MATLAB displays information about the object:

```
hobj1 =
```

```
  HdClass handle
  Properties:
    Data: 8
```

The variable `hobj1` is a handle that references the object created. Copying `hobj1` to `hobj2` results in another handle (the variable `hobj2`) referring to the same object:

```
hobj2 = hobj1
hobj2 =
  HdClass handle
  Properties:
    Data: 8
```

Because handle objects reference the data contained in their properties, copying an object copies the handle to a new variable name, but the properties still refer to the same data. For example, given that `hobj1` is a handle object with property `Data`:

```
hobj1.Data
ans =
     8
```

When you change the value of `hobj1`'s `Data` property, the value of the copied object's `Data` property also changes:

```
hobj1.Data = 5;

hobj2.Data
ans =
     5
```

Because `hobj2` and `hobj1` are handles to the same object, changing the copy, `hobj2`, also changes the data you access through handle `hobj1`:

```
hobj2.Data = 17;
hobj1.Data
ans =
    17
```

Reassigning Handle Variables

Reassigning a handle variable produces the same result as reassigning any MATLAB variable. When you create a new object and assign it to `hobj1`:

```
hobj1 = HdClass(3.14);
```

`hobj1` references the new object, not the same object referenced previously (and still referenced by `hobj2`).

Clearing Handle Variables

When you `clear` a handle from the workspace, MATLAB removes the variable, but does not removed the object referenced by the handle. Therefore, given `hobj1` and `hobj2`, which both reference the same object, you can clear either handle without affecting the object:

```
hobj1.Data = 2^8;
clear hobj1
hobj2
hobj2 =
  HdClass handle
  Properties:
    Data: 256
```

If you clear both `hobj1` and `hobj2`, then there are no references to the object and MATLAB deletes the object and frees the memory used by that object.

Deleting Handle Objects

To remove an object referenced by any number of handles, you `delete` the object. Given `hobj1` and `hobj2`, which both reference the same object, if you delete either handle, MATLAB deletes the object:

```
hobj1 = HdClass(8);
hobj2 = hobj1;
delete(hobj1)
hobj2
hobj2 =
  deleted HdClass handle
```

See "Destroying Objects" on page 5-31 for more information about object lifecycle.

Testing for Handle or Value Class

If you are writing MATLAB programs that copy objects, you might need to determine if any given object is a handle or value. To determine if an object is a handle object, use the `isa` function:

```
isa(obj,'handle')
```

For example, the `containers.Map` class creates a handle object:

```
hobj = containers.Map({'Red Sox','Yankees'}, {'Boston','New York'});
isa(hobj,'handle')
ans =
     1
```

`hobj` is also a `containers.Map` object:

```
isa(hobj,'containers.Map')
ans =
     1
```

If you query the `class` of `hobj`, you see that it is a `containers.Map` object:

```
class(hobj)
ans =
containers.Map
```

The `class` function returns the specific class of an object, whereas `isa` returns `true` for any of the object's superclasses as well. This behavior is consistent with the object-oriented concept that an object is a member of all its superclasses. Therefore, it is true that a `containers.Map` object is a handle object and a `containers.Map` object.

There is no equivalent test for value classes because there is no value base class. If an object is a value object, `isa(object,'handle')` returns `false` (i.e., logical `0`).

See "Map Containers" on page 1-144 for more information on the `containers.Map` class.

Destroying Objects

In this section...

Object Lifecycle

An object's lifecycle ends when you reassign a new value to that variable, when it is no longer used in a function, or when function execution ends. MATLAB handle classes have a special method called `delete` that MATLAB calls when a handle object lifecycle ends.

Calling `delete` on an object explicitly makes all copies of a handle object invalid because it destroys the data associated with the object and frees memory used by deleted objects. MATLAB calls `delete` automatically so it is not necessary for you to do so. Classes can redefine the `handle` class `delete` method to perform other cleanup operations, like closing files or saving data.

Deleting a `handle` object renders all copies invalid:

```
delete(hobj1)
hobj2.a
??? Invalid or deleted object.
```

Difference Between clear and delete

The `handle` class `delete` method removes the `handle` object, but does not clear the variable name. The `clear` function removes a variable name, but does not remove the values to which the variable refers. For example, if you have two variables that refer to the same handle object, you can clear either one without affecting the actual object:

```
hobj = containers.Map({'Red Sox','Yankees'}, {'Boston','New York'});
hobj_copy = hobj;
clear hobj
city = hobj_copy('Red Sox')
city =
Boston
```

If you call `delete` on all handle variables that refer to the same handle object, then you have lost access to the object and MATLAB destroys the object. That is, when there are no references to an object, the object ceases to exist.

If you call `delete` on a value object, MATLAB returns an error. You can only call `clear` on value objects.

Data Import and Export

Overview

In this section...

For more information and examples on importing and exporting data, see Technical Note 1602:

`http://www.mathworks.com/support/tech-notes/1600/1602.html`

Supported File Types

The MATLAB software provides many ways to load data from disk files or the clipboard into the workspace, a process called *importing* data. Also there are many ways to save workspace variables to a disk file, a process called *exporting* data. Your choice of which import or export mechanism to use depends mainly on the format of the data being transferred: text, binary, or a standard format such as JPEG.

Note For unsupported high-level function data formats, you can use the MATLAB low-level file I/O functions if you know how the binary data is formatted in the file. For more information, see "Using Low-Level File I/O Functions" on page 6-67.

MATLAB has built-in capabilities to import and export the following types of data files:

- “Data from the System Clipboard” on page 6-4
- “Information from the Internet” on page 6-4

Binary Data from a MATLAB Session

Using the MATLAB `save` and `load` functions, you can store all or part of the data in your MATLAB workspaces to disk, and then read that data back into MATLAB at a later time.

Text Data

In text format, the data values are American Standard Code for Information Interchange (ASCII) codes that represent alphabetic and numeric characters. You can view ASCII text data in a text editor. For more information about working with text data in MATLAB, see these sections:

- “Importing Text Data” on page 6-35
- “Exporting Text Data” on page 6-44

They also describe how to import and export to XML documents.

Graphics Files

MATLAB imports and exports images from many standard graphics file formats, including the Tagged Image File Format (TIFF), Graphics Interchange Format (GIF), Joint Photographic Experts Group (JPEG), and Portable Network Graphics (PNG) formats.

Audio and Audio/Video Data

MATLAB provides functions to enable you to interact with the following types of audio and audio/video files:

- NeXT/Sun SPARC®station sound
- Microsoft WAVE sound
- Audio/Video Interleaved (AVI)
- Sound devices compatible with Microsoft Windows
- Audio player and recorder objects

- Linear audio signals

Spreadsheets

You can use MATLAB to import and export data to the following types of spreadsheets:

- Microsoft® Excel® spreadsheets
- Lotus 123 spreadsheets

Data from the System Clipboard

Using the Import Wizard or the `clipboard` function, you can temporarily hold string data on your system's clipboard, and then paste it back into MATLAB.

Information from the Internet

From your MATLAB session, you can

- Send e-mail
- Download from the Internet
- Compress (zip) and decompress (unzip) files
- Connect to an FTP server to perform remote file operations

Other MATLAB I/O Capabilities

- "Using the Import Wizard" on page 6-4
- "Mapping Files to Memory" on page 6-5
- "Low-Level File I/O" on page 6-5
- "Importing Data with Toolboxes" on page 6-5

Using the Import Wizard

The Import Wizard is a graphical user interface that simplifies the process of locating and loading various types of data files into MATLAB. You do not need to know the format of the data to use this tool. You simply specify the

file that contains the data and the Import Wizard processes the file contents automatically. See the section on “Using the Import Wizard” on page 6-11.

Mapping Files to Memory

Memory-mapping enables you to read and write data in a file as if were stored in the computer’s dynamic memory. The contents of the mapped file appear as if they were an array in the currently active workspace. You simply index into this array to read or write the desired data from the file. See the section on “Accessing Files with Memory-Mapping” on page 6-80.

Low-Level File I/O

MATLAB also supports C-style, low-level I/O functions that you can use with any data format. For more information, see “Using Low-Level File I/O Functions” on page 6-67.

Importing Data with Toolboxes

In addition to MATLAB import functions, you can perform specialized import features using toolboxes. For example, use Database Toolbox™ software for importing data from relational databases. Refer to the documentation on the specific toolbox to see what import features are offered.

Functions Used in File Management

The following functions are available in MATLAB to help you to create, manage, and locate the files and directories you work with. For more information on these and other file management functions, see “Managing Files and Working with the Current Directory” in the MATLAB Desktop Tools and Development Environment documentation:

Function	Description
`cd`	Switch your current working directory to another directory
`clipboard`	Copy and paste strings to and from the system clipboard
`copyfile`	Copy a file or directory to another location
`delete`	Delete the specified files
`dir`	List the files that reside in the specified directory

Function	Description
`edit`	Create a new M-file or edit an existing one
`exist`	Check the existence of a file or directory
`fileattrib`	Set or get attributes of a file or directory
`filebrowser`	Start the Current Directory Browser
`fileparts`	Show the components of a file name and its place on the path
`fullfile`	Build a full file name from its components
`ls`	List the contents of a specific directory
`mkdir`	Create a new directory
`movefile`	Move a file or directory to a new location
`open`	Open files based on extension
`pwd`	Identify the directory you are currently working in
`recycle`	Set an option to move deleted files to recycle folder
`rmdir`	Delete a specific directory
`what`	List the MATLAB files in a specific directory
`which`	Locate functions and files

Supported File Formats

The table below shows the file formats that you can read or write from your MATLAB application, along with the functions that support each format.

File Content	Extension	Description	Import Function	Export Function
MATLAB formatted data	MAT	Saved MATLAB workspace	`load`	`save`
Text	any	White-space delimited numbers	`load`	`save -ascii`
		Delimited numbers	`dlmread`	`dlmwrite`
		Comma delimited numbers	`csvread`	`csvwrite`
		Any of the above text formats, or a mix of strings and numbers	`textscan`	
Spreadsheet	XLS	Microsoft Excel worksheet	`xlsread`	`xlswrite`
	XLSX XLSB XLSM	Formats supported with Excel® 2007 for Windows installed		
Extended Markup Language	XML	XML-formatted text	`xmlread`	`xmlwrite`
Data Acquisition Toolbox™ file	DAQ	Data Acquisition Toolbox	`daqread`	none

File Content	Extension	Description	Import Function	Export Function
Scientific data	CDF	Common Data Format	`cdfread`	`cdfwrite`
	FITS	Flexible Image Transport System	`fitsread`	none
	HDF	Hierarchical Data Format, version 4, or HDF-EOS v. 2	`hdfread`	
	H5	HDF or HDF-EOS, version 5	`hdf5read`	`hdf5write`
	NC	Network Common Data Form (netCDF)	See `netcdf`	See `netcdf`
Image	BMP	Windows Bitmap	`imread`	`imwrite`
	GIF	Graphics Interchange Format		
	HDF	Hierarchical Data Format		
	JPEG JPG	Joint Photographic Experts Group		
	PBM	Portable Bitmap		
	PCX	Paintbrush		
	PGM	Portable Graymap		
	PNG	Portable Network Graphics		
	PNM	Portable Any Map		
	PPM	Portable Pixmap		
	RAS	Sun Raster		
	TIFF TIF	Tagged Image File Format		
	XWD	X Window Dump		

File Content	Extension	Description	Import Function	Export Function
	CUR	Windows Cursor resources	`imread`	none
	FITS FTS	Flexible Image Transport System		
	ICO	Windows Icon resources		
	JP2 JPF JPX J2C J2K	JPEG 2000		
Audio file	AU SND	NeXT/Sun sound	`auread`	`auwrite`
	WAV	Microsoft WAVE sound	`wavread`	`wavwrite`
Video (all platforms)	AVI	Audio Video Interleave	`aviread`, `mmreader`	`avifile`
Video (Windows)	MPG	Motion Picture Experts Group, phases 1 and 2	`mmreader`	none
	ASF ASX WMV	Windows Media®		
	any	Formats supported by Microsoft DirectShow®		

File Content	Extension	Description	Import Function	Export Function
Video (Mac®)	MPG MP4 M4V	MPEG-1 and MPEG-4	`mmreader`	none
	any	Formats supported by QuickTime®, including `.mov`, `.3gp`, `.3g2`, and `.dv`		
Video (Linux®)	any	Formats supported by your installed GStreamer plug-ins, including `.ogg`	`mmreader`	none

Using the Import Wizard

In this section...

Overview

The easiest way to import data into your MATLAB application is to use the Import Wizard. You do not need to know the format of the data to use this tool. You simply specify the file that contains the data and the Import Wizard processes the file contents automatically. You can also use the Import Wizard to import HDF data. For more information, see "Using the HDF Import Tool" on page 7-45.

The sections on Previewing Contents of the File or Clipboard and Specifying Delimiters and Header Format apply only to text files and the clipboard.

Starting the Import Wizard

To start the Import Wizard and select the source to import, see these sections:

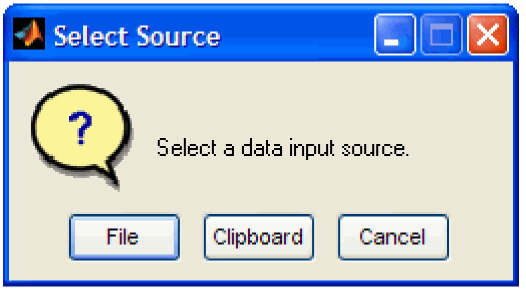
- "Importing from a File" on page 6-12
- "Importing from the Clipboard" on page 6-12

If you use the `uiimport` function to start the Wizard, you can choose to have the imported data written to a MATLAB structure. See "Importing to a Structure" on page 6-13.

Importing from a File

To start the Wizard and use a file browser to locate the file to import, use one of the menu options or MATLAB commands shown here:

- Select **Import Data** from the **File** menu
- Type `uiimport -file`, and press **Enter**
- Type `uiimport`, press **Enter**, and click **File** in the **Import Data** dialog box



If you already know the name of the file to import, use one of the following means to initiate the operation:

- In the Current Directory browser, right-click the filename and select **Import Data**
- Type `uiimport filename`, where `filename` is an unquoted string containing the name of the file to import.

Importing from the Clipboard

To import from the system clipboard, use one of the menu options or MATLAB commands shown here:

- Select **Paste to Workspace** from the **Edit** menu
- Type `uiimport -pastespecial`, and press **Enter**
- Type `uiimport`, press **Enter**, and click **Clipboard** in the **Import Data** dialog box

Importing to a Structure

Specifying an output argument with the `uiimport` command tells MATLAB to return the imported data in the fields of a single structure rather than as separate variables.

The command

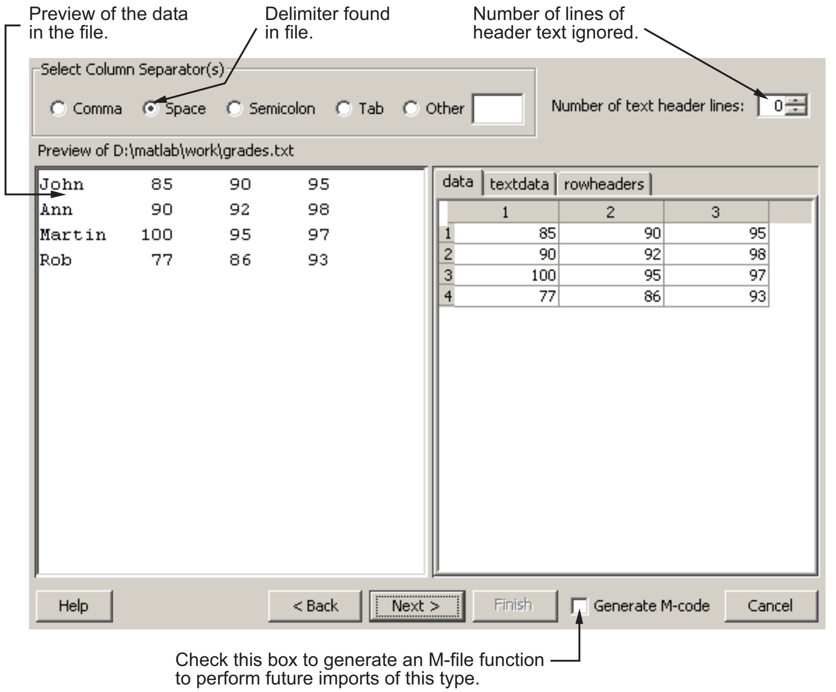
```
S = uiimport('filename')
```

imports the file `filename` to the fields of structure `S`. The `filename` argument is a single-quoted string containing the name of the file to import.

> If you are importing from a binary file, skip ahead Determine Assignment to Variables.

Previewing Contents of the File or Clipboard [Text only]

When the Import Wizard imports text data from a file or the clipboard, it opens the dialog box shown here and displays a portion of the raw data in the preview pane on the left. You can use this display to verify that the file contains the data you expect.



The pane on the right side of the dialog box shows how MATLAB has assigned the imported data to a default set of variables. The variable names appear in the tabs above the display pane. Click any of these tabs to see the values assigned to that variable. The variable names are derived from categories into which the Import Wizard has sorted the data. These are as follows:

- `rowheaders` — Column vector containing the names of all row headers.
- `colheaders` — Row vector containing the names of all column headers.
- `textdata` — Matrix containing all imported text data. Empty elements are set to `''`.
- `data` — Matrix containing all imported numeric data. Empty elements are set to `NaN`.

If the imported file or clipboard contains *only* numeric or *only* text data, then the Import Wizard does not use the variable names shown above. Instead, it assigns all of the data to just one variable:

- For data imported from a text file, the name of the variable is the same as the filename, minus the file extension.
- For data imported from the clipboard, the name of the variable is `A_pastespecial`.

Specifying Delimiters and Header Format [Text only]

Using the options shown at the top of the Import Wizard dialog box, you can specify a delimiter character for separating data items, and also the number of lines you want to use for column headers.

Delimiters

Most text files use a unique character called a *delimiter* or column separator to mark the separation between items of data. For example, data in a comma-separated value (CSV) file is, of course, separated by commas. Data in some other file might be separated by tab or space characters.

When the Import Wizard imports from a text file or the clipboard, it makes its best guess as to which character was intended as the delimiter and displays the data accordingly. If this is not correct, you will need to set the correct delimiter from the choices shown under **Select Column Separator(s)** in the upper left of the dialog box. When you do this, the Import Wizard immediately reformats the data, displaying new values for the data shown in the preview pane.

Header Format

When reading in data from a text file or the clipboard, the Wizard looks for any lines at the top that have no numeric characters, and assigns these lines to the variable `textdata`. MATLAB counts these lines and displays the count in the **Number of text header lines** value field in the upper right of the Import Wizard window. You can adjust this count if it does not accurately represent the header format within the file.

Note The **Number of text header lines** selector applies only to column headers. It has no effect on row headers.

MATLAB creates a row vector from the bottommost of these lines and assigns it to the variable `colheaders`.

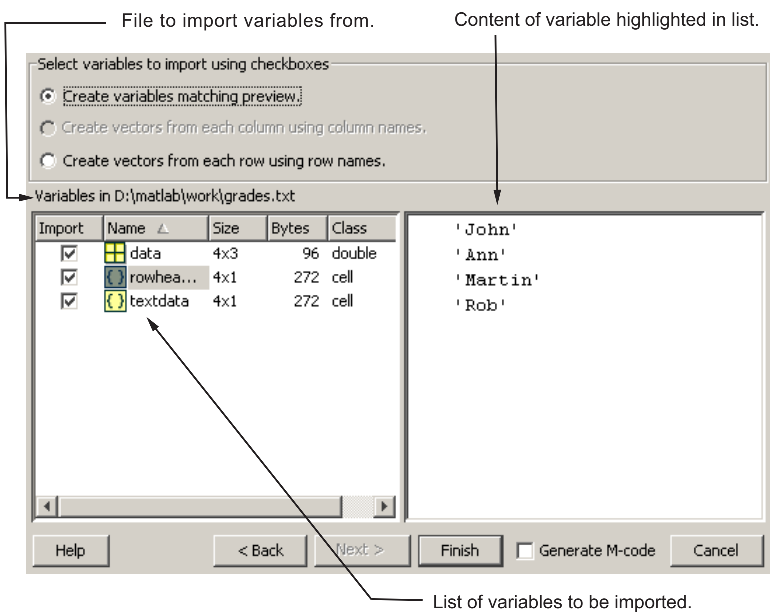
Generate M-Code Checkbox

The **Generate M-code** checkbox at the bottom of the Import Wizard dialog box applies to both text and binary data, and thus is described in “Automated M-Code Generation” on page 6-19.

To continue, click **Next** at the bottom of the dialog box.

Determining Assignment to Variables

At this point, the Import Wizard displays the dialog box shown below. This dialog displays data for both text and binary files.



The left pane of the dialog box displays a list of the variables MATLAB created for your data. For text files, MATLAB derives the variable names as described in Preview Contents of the File. For binary files, the variable names are taken directly from the file.

Click any variable name and MATLAB displays the contents of that variable in the pane to the right. MATLAB highlights the name of the variable that is currently displayed in the right pane.

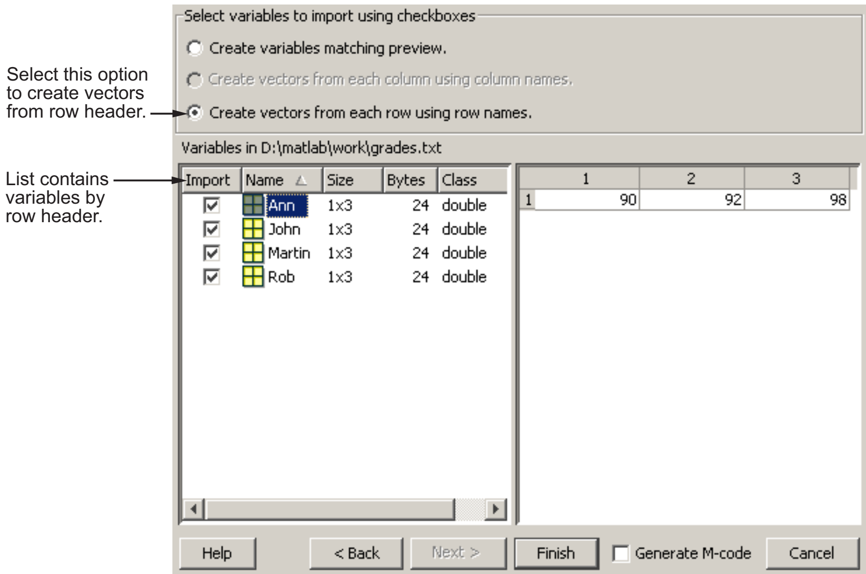
Structuring the Output Data

The top portion of this dialog box offers three options for organizing the file's data:

- **Create variables matching preview**
- **Create vectors from each column using column names**
- **Create vectors from each row using row names**

Note For data imported from a binary file, only the top option is active. Variable names and assignment are taken directly from the imported file. For text data, you can use any of the three options; however, the bottom two are active only if the file or clipboard contains row or column headers.

While importing from the example text file `grades.txt`, select the third option to create vectors from row names. Observe that the display replaces the default variable assignments with new variables derived from the row headers. Click any of these variable names, and the Wizard displays the contents of the corresponding row vector.



Selecting Which Variables to Write to the Workspace

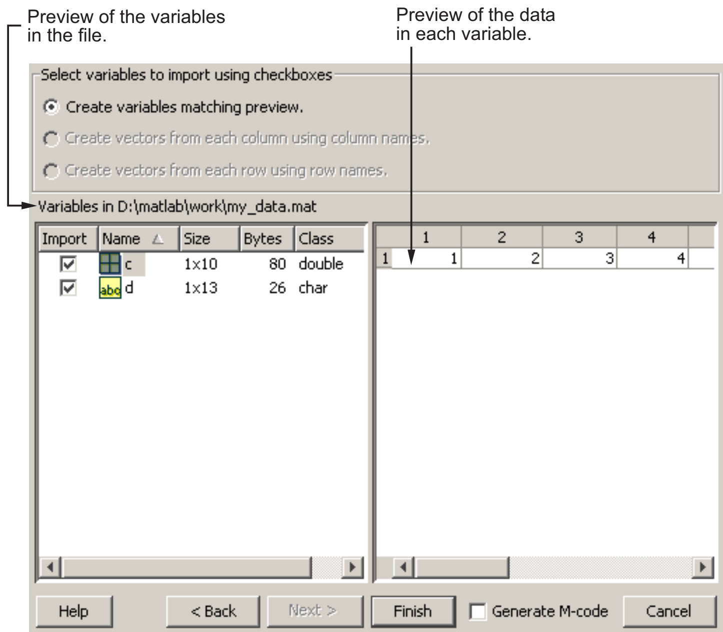
The checkboxes to the left of each variable name enable you to include or exclude individual variables from those that will be written to the workspace. By default, all variables are selected. Select the checkbox of any variable you do not want written to the workspace. The check mark is removed from any variables that you deselect.

Example of Selecting Variables to Load. Use the Import Wizard to import this sample binary MAT-file, `my_data.mat`,

```
C =
     1  2  3  4  5
     6  7  8  9  10

D =
     a test string
```

The Import Wizard displays two variables, as listed in the preview pane. To select a variable to import, select the check box next to its name. All variables are preselected by default.



Automated M-Code Generation

To perform additional imports from this or a similar type of file, you can automate this process by creating a MATLAB function that performs all of the steps you just went through. To have the Import Wizard write this function for you, select the **Generate M-code** checkbox in the lower right corner of the Wizard dialog.

Once you click **Finish** to complete the import, MATLAB opens an Editor window displaying the generated M-file function. The function is called `importfile.m`. If this name is already taken, then MATLAB names the file `importfileN.m`, where `N` is a number that is one greater than the highest existing `importfile.m` file.

The generated function has the following input and output arguments:

- Input: `fileToRead1` — Name of the file to import from. This argument exists only when importing from a file.

- Output: `newData1` — Structure to assign all imported data to. This argument exists only if you have specified an output argument with the call to `uiimport` when starting the Import Wizard. Otherwise, variables retain the same naming as assigned within the Wizard.

The `newData1` output is a structure that has one field for each output of the import operation.

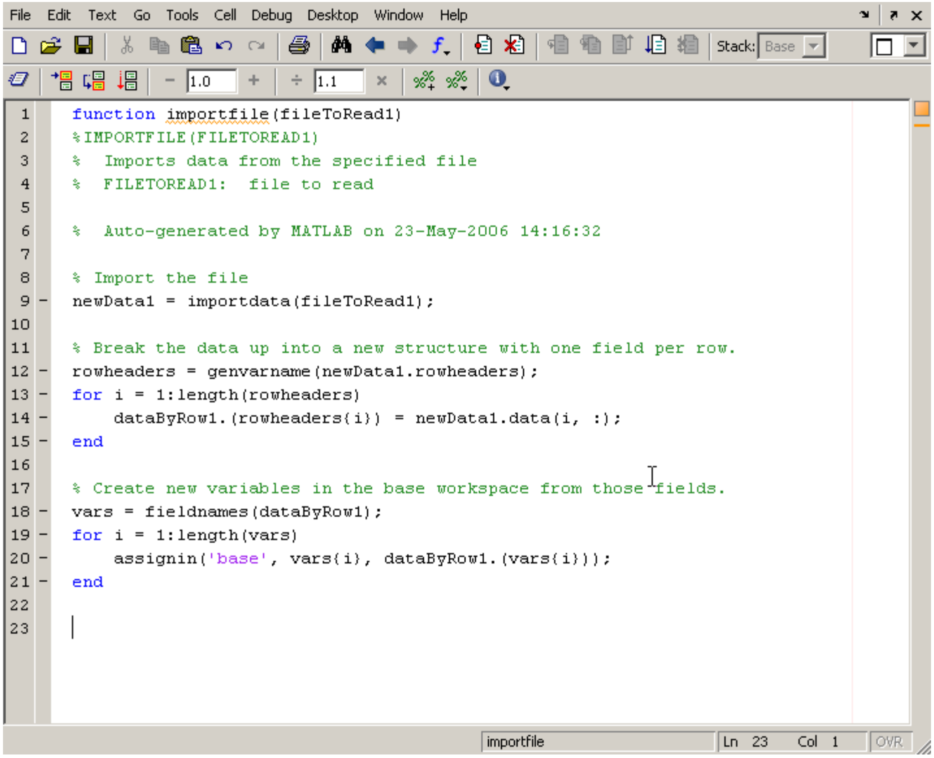
The workspace variables created by this generated M-code are the same as those created by running the Import Wizard. For example, if you elect to format the output in column vectors when running the Import Wizard, the generated M-file does the same. However, unlike the Import Wizard, you cannot mark any variables to be excluded from the output.

Make any necessary modifications to the generated M-file function in the Editor window. To save the M-file, select **Save** from the **File** menu at the top.

Caution You must save the file yourself; MATLAB does not automatically save it for you.

Example of M-Code Generation

The M-file shown below was generated by MATLAB during an import of the file `grades.txt`, shown earlier in this section. During the import that created this file, the option to **Create vectors from each row using row names** was selected, thus generating four row vectors for output: `Ann`, `John`, `Martin`, and `Rob`. Also, the row vector for `John` was deselected by clearing the checkbox next to that name in the Wizard.



If you run the function, you find that the workspace now holds the four row vectors `Ann`, `John`, `Martin`, and `Rob`, instead of the default variables created by the Import Wizard (`data`, `textdata`, and `rowheaders`). Also, note that the vector for `John` is written to the workspace along with the others, even though this one variable had been deselected from the Import Wizard interface.

```
importfile grades.txt

whos
  Name          Size             Bytes  Class     Attributes

  Ann           1x3                 24  double
  John          1x3                 24  double
  Martin        1x3                 24  double
  Rob           1x3                 24  double
```

Writing Data to the Workspace

To complete the import operation, click **Finish** to bring the data into the MATLAB workspace. This button also dismisses the Import Wizard.

Variables written to the workspace are in one of the following formats. The first three apply only to data read from text files or the clipboard, the fourth applies only to binary files, and the last applies to both:

Variable Name	Output
`data`, `textdata`, `rowheaders`, `colheaders`	Separate matrices for numeric, text, and header data.
Variables named after row or column headers	One vector for each row or column.
Single variable named after the file name, or `A_pastespecial`	One matrix for all data named after the file name
Variable names taken from binary file	Data assigned to each variable stored in a binary file.
Output variable assigned during call to `uiimport`	A single structure having fields that match one of the formats described above.

Examples

Here are a few examples of how to use the Import Wizard.

Example 1—Text Data. Start by creating the text file `grades.txt` using the MATLAB editor. The file contains the following:

```
John      85      90      95
Ann       90      92      98
Martin   100      95      97
Rob       77      86      93
```

Import from text file `grades.txt`, using default variables to store the data:

```
uiimport grades.txt
```

```
whos
  Name              Size              Bytes  Class     Attributes

  data              4x3                  96  double
  rowheaders        4x1                 272  cell
  textdata          4x1                 272  cell
```

Example 2— Partial Text File with Row Vectors. Import from the same file as in the above example, but this time select **Create vectors from each row using row names**. Also, clear the checkbox next to variable `John` so that this one vector does not get written to the workspace:

```
whos
  Name          Size              Bytes  Class     Attributes

  Ann           1x3                  24  double
  Martin        1x3                  24  double
  Rob           1x3                  24  double
```

Example 3 — Binary Data Assigned to a Structure. Save numeric and text data in binary format in file `importtest.mat` and use the Import Wizard to import the binary file into the workspace.

```
C = [1 2 3 4 5;6 7 8 9 10];
D = 'a test string';
save importtest C D

clear
s = uiimport('importtest.mat')
s =
    C: [2x5 double]
    D: 'a test string'
```

Exporting Data to MAT-Files

In this section...

MAT-Files

MAT-files are double-precision, binary, MATLAB format files. They can be created on one machine and later read by MATLAB on another machine with a different floating-point format, retaining as much accuracy and range as the different formats allow. They can also be manipulated by other programs external to MATLAB.

This section explains how to save the variables in your MATLAB session to a binary file called a MAT-file. The next section explains how to load the variables back into your MATLAB workspace.

Using the save Function

To export workspace variables to a binary or ASCII file, use the `save` function. You can save all variables from the workspace in a single operation (if you omit the filename, MATLAB uses the name `matlab.mat)`:

```
save filename
```

or save just those variables that you specify:

```
save filename var1 var2 ... varN
```

Use the wildcard character (*) in the variable name to save those variables that match a specific pattern. For example, the following command saves all variables that start with `str`:

```
save strinfo str*
```

Use `whos -file` to examine what has been written to the MAT-file:

```
whos -file strinfo
  Name          Size                    Bytes  Class

  str2          1x15                       30  char array
  strarray      2x5                       678  cell array
  strlen        1x1                         8  double array
```

Saving Structures

When saving a MATLAB structure, you have the option of saving the entire structure, saving each structure field as an individual variable in the MAT-file, or saving specific fields as individual variables.

For structure `S`:

```
S.a = 12.7;  S.b = {'abc', [4 5; 6 7]};  S.c = 'Hello!';
```

Save the entire structure to `newstruct.mat` with the usual syntax:

```
save newstruct.mat S;

whos -file newstruct
  Name      Size                    Bytes  Class

  S         1x1                       550  struct array
```

Save the fields individually with the `-struct` option:

```
save newstruct.mat -struct S;

whos -file newstruct
```

```
  Name      Size                    Bytes  Class

  a         1x1                         8  double array
  b         1x2                       158  cell array
  c         1x6                        12  char array
```

Or save only selected fields using `-struct` and specifying each field name:

```
save newstruct.mat -struct S a c;

whos -file newstruct
  Name      Size                    Bytes  Class

  a         1x1                         8  double array
  c         1x6                        12  char array
```

Appending to an Existing File

You can add new variables to those already stored in an existing MAT-file by using `save -append`. When you append to a MAT-file, MATLAB first looks in the designated file for each variable name specified in the argument list, or for all variables if no specific variable names are specified. Based on that information, MATLAB does both of the following:

- For each variable that already exists in the MAT-file, MATLAB overwrites its saved value with the new value taken from the workspace.
- For each variable not found in the MAT-file, MATLAB adds that variable to the file and stores its value from the workspace.

Note Saving with the `-append` option does not append additional elements to any arrays that are already saved in the MAT-file.

Data Compression

MATLAB compresses the data that you save to a MAT-file. Data compression can save you a significant amount of storage space when you are working with large files or working over a network.

Data compression is optional, however, and you can disable it either for an individual `save` operation, or for all of your MATLAB sessions. Use the `-v6` option with the `save` function to turn off compression on a per-command basis:

```
save filename -v6
```

To disable data compression for all of your MATLAB sessions, open the **Preferences** dialog, select **General**, and then select **MAT-Files**. Next click the option that is equivalent to the command `save -v6`. See General Preferences for MATLAB in the Desktop Tools and Development Environment documentation for more information.

Note You cannot read a compressed MAT-file with MATLAB software versions earlier than Version 7. To write a MAT-file that you will be able to read with one of these versions, save to the file with data compression disabled.

Information returned by the command `whos -file` is independent of whether the variables in that file are compressed or not. The byte counts returned by this command represent the number of bytes data occupies in the MATLAB workspace, and not in the file the data was saved to.

Evaluating When to Compress

You should consider both data set size and the type of data being saved when deciding whether or not to compress the data you save to a file. The benefits of data compression are greater when saving large data sets (over 3 megabytes), and are usually negligible with smaller data sets. Data that has repeating patterns or more consistent values compresses better than random data. Compressing data that has a random pattern is not recommended as it slows down the performance of `save` and `load` significantly, and offers little benefit in return.

In general, data compression and decompression slows down all `save` and some `load` operations to some extent. In most cases, however, the resulting reduction in file size is worth the additional time spent compressing or decompressing. Because loading is typically done more frequently than saving, `load` is considered to be the most critical of the two operations. Up to a certain threshold (relative to the size of the uncompressed MAT-file), loading

a compressed MAT-File is slightly slower than loading an uncompressed file containing the same data. Beyond that threshold, however, loading the compressed file is *faster*.

For example, say that you have a block of data that takes up 100 megabytes in memory, and this data has been saved to both a 10 megabytes compressed file and a 100 megabytes uncompressed file. When you load each of these files back into the MATLAB workspace, the first 10 megabytes of data takes the same amount of time to load for each file. Loading the remaining 90 megabytes from the uncompressed file will take nine times as long as the first 10 megabytes, while all that remains to be done with the compressed file is to decompress the data, and this takes a relatively short amount of time.

The loading size threshold is lower for network files, and also varies depending on the type of computer being used. Network users loading compressed MAT-files generally see faster load times than when loading uncompressed files, and at smaller data sizes than users loading the same files locally.

Note Compression and decompression during `save` and `load` is done transparently without the use of temporary files on disk. This is of significance to large dataset users in particular.

Unicode Character Encoding

MATLAB saves character data to a MAT-file using the Unicode, Inc. Unicode character encoding. As with data compression, Unicode character encoding is optional. If you disable it, MATLAB writes the MAT-file using the default encoding for your system. To disable Unicode character encoding on a per-command basis, use the `-v6` option with the `save` function:

```
save filename -v6
```

To disable Unicode character encoding for all of your MATLAB sessions, open the **Preferences** dialog box, select **General**, and then select **MAT-Files**. Next click the option that is equivalent to the command `save -v6`. See General Preferences for MATLAB in the Desktop Tools and Development Environment documentation for more information.

When writing character data to a non-HDF5-based MAT-file using Unicode encoding (the default), MATLAB checks if the data is 7-bit ASCII. If it is, MATLAB writes the 7-bit ASCII character data to the MAT-file using 8 bits per character (UTF-8 format), thus minimizing the size of the resulting file. Any character data that is not 7-bit ASCII is written in 16-bit Unicode form (UTF-16). This algorithm operates on a per-string basis.

Note You cannot read a Unicode encoded MAT-file with MATLAB versions earlier than Version 7. To write a MAT-file that you will be able to read with one of these versions, save to the file with Unicode character encoding disabled.

For more information on how MATLAB saves specific ASCII data formats, and on preventing loss or corruption of character data, see "Writing Character Data" in the MATLAB External Interfaces documentation.

Optional Output Formats

You can choose from any of the following formats for your output file. If you do not specify a format, MATLAB uses the binary MAT-file format.

Output File Format	Command
Binary MAT-file (default)	`save` *`filename`*
8-digit ASCII	`save` *`filename`* `-ascii`
8-digit ASCII, tab delimited	`save` *`filename`* `-ascii -tabs`
16-digit ASCII	`save` *`filename`* `-ascii -double`
16-digit ASCII, tab delimited	`save` *`filename`* `-ascii -double -tabs`
MATLAB Version 4 compatible	`save` *`filename`* `-v4`

Saving in ASCII Format

When saving in any of the ASCII formats, consider the following:

- Each variable to be saved must be either a two-dimensional double array or a two-dimensional character array. Saving a complex double array

causes the imaginary part of the data to be lost, as MATLAB cannot load nonnumeric data ('i').

- To read the file with the MATLAB `load` function, make sure all the variables have the same number of columns. If you are using a program other than MATLAB to read the saved data, this restriction can be relaxed.
- Each MATLAB character in a character array is converted to a floating-point number equal to its internal ASCII code and written out as a floating-point number string. There is no information in the saved file that indicates whether the value was originally a number or a character.
- The values of all variables saved merge into a single variable that takes the name of the ASCII file (minus any extension). Therefore, it is advisable to save only one variable at a time.

Saving in Version 4 Format

With the `-v4` option, you can save only those data constructs that are compatible with MATLAB Version 4. Therefore, you cannot save structures, cell arrays, multidimensional arrays, or objects. Variable names cannot exceed 19 characters in length. In addition, you must use filenames that are supported by MATLAB Version 4.

Storage Requirements

The binary formats used by `save` depend on the size and type of each array. Arrays with any noninteger entries and arrays with 10,000 or fewer elements are saved in floating-point formats requiring 8 bytes per real element. Arrays with all integer entries and more than 10,000 elements are saved in the formats shown, requiring fewer bytes per element.

Element Range	Bytes per Element
0 to 255	1
0 to 65535	2
-32767 to 32767	2
-2^{31} to $2^{31}-1$	4
Other	8

Saving from External Programs

The MATLAB External Interfaces documentation provides details on reading and writing MAT-files from external C or Fortran programs. It is important to use recommended access methods, rather than rely upon the specific MAT-file format, which is likely to change in the future.

Importing Data From MAT-Files

In this section...

Using the load Function

To import variables from a binary or ASCII file on your disk to your workspace, use the `load` function. You can load all variables from the workspace in a single operation (if you omit the filename, the MATLAB software loads from file `matlab.mat`):

```
load filename
```

or load just those variables that you specify:

```
load filename var1 var2 ... varN
```

Use the wildcard character (*) in the variable name to load those variables that match a specific pattern. (This works for MAT-files only.) For example, the following command loads all variables that start with `str` from file `strinfo.mat`:

```
load strinfo str*
```

Caution When you import data into the MATLAB workspace, it overwrites any existing variable in the workspace with the same name.

Previewing MAT-File Contents

To see what variables are stored in a MAT-file before actually loading the file into your workspace, use `whos -file` *`filename`*. This command returns the name, dimensions, size, and class of all variables in the specified MAT-file.

You can use `whos -file` on binary MAT-files only:

```
whos -file mydata.mat
  Name           Size             Bytes  Class

  javArray      10x1                       java.lang.Double[][]
  spArray        5x5                 84  double array (sparse)
  strArray       2x5                678  cell array
  x              3x2x2               96  double array
  y              4x5               1230  cell array
```

Loading Into a Structure

To load MAT-file data into a MATLAB structure, specify an output variable in your `load` command. This example reads the data in `mydata.mat` into the fields of structure `S`:

```
S = load('mydata.mat')
S =
           x: [3x2x2 double]
           y: {4x5 cell}
     spArray: [5x5 double]
    strArray: {2x5 cell}
    javArray: [10x1 java.lang.Double[][]]

whos S
  Name      Size                       Bytes  Class

  S         1x1                         2840  struct array
```

Loading Binary Data

MAT-files are double-precision binary MATLAB format files created by the `save` function and readable by the `load` function. You can create MAT-files on one machine and later MATLAB can read them on another machine with a different floating-point format, retaining as much accuracy and range as the different formats allow. Other programs, external to MATLAB, can also manipulate MAT-files.

MAT-files can contain data in an uncompressed or a compressed form, or both. MATLAB knows which variables in the file have been compressed by looking at a tag that it attaches to each variable during the `save` operation. When

loading data from a MAT-file into the workspace, MATLAB automatically handles the decompression of the appropriate data.

The External Interface libraries contain C- and Fortran-callable routines to read and write MAT-files from external programs.

Loading ASCII Data

To use the `load` function, ASCII files must be organized as a rectangular table of numbers, with each number in a row separated by a blank, comma, semicolon, or tab character, and with an equal number of elements in each row. MATLAB generates an error if the number of values differs between any two rows. ASCII files can contain MATLAB comments (lines that begin with `%`).

MATLAB returns all the data in the file as a single two-dimensional array of type `double`. The number of rows in the array is equal to the number of lines in the file, and the number of columns is equal to the number of values on a line.

In the workspace, MATLAB assigns the array to a variable named after the file being loaded (minus any file extension). For example, the command

```
load mydata.dat
```

reads all of the data from `mydata.dat` into the MATLAB workspace as a single array, and assigns it to a variable called `mydata`. In naming the variable, `load` precedes any leading underscores or digits in `filename` with an `X` and replaces any other nonalphabetic characters with underscores.

For example, the command

```
load 10-May-data.dat
```

assigns the data in file `10-May-data.dat` to a new workspace variable called `X10_May_data`.

Importing Text Data

In this section...

Caution When you import data into the MATLAB workspace, you overwrite any existing variable in the workspace with the same name.

The MATLAB Import Wizard

The easiest way to import data into your MATLAB application is to use the Import Wizard, a graphical user interface. The Import Wizard automatically reads numeric data in any text file you specify. The Import Wizard also recognizes text strings as row and column headers in your data.

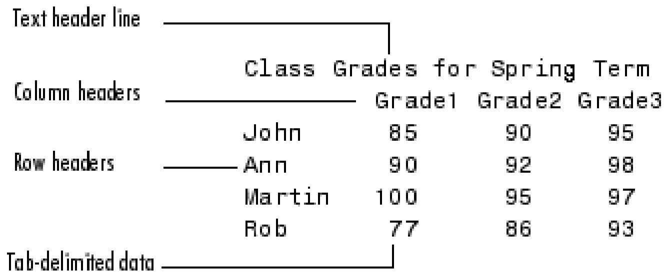
For more information, see "Using the Import Wizard" on page 6-11.

Using Import Functions with Text Data

To import text data from the command line or in an M-file, you must use one of the MATLAB import functions. Your choice of function depends on how the data in the text file is formatted.

The text data must be formatted in a uniform pattern of rows and columns, using a text character, called a *delimiter* or *column separator*, to separate each data item. The delimiter can be a space, comma, semicolon, tab, or any other character. The individual data items can be alphabetic or numeric characters or a mix of both.

The text file can also contain one or more lines of text, called *header lines*, or can use text headers to label each column or row. The following example illustrates a tab-delimited text file with header text and row and column headers.



To find out how your data is formatted, view it in a text editor. After you determine the format, find the sample in the table below that most closely resembles the format of your data. Then read the topic referred to in the table for information on how to import that format.

Table 6-1 ASCII Data File Formats

Data Format Sample	Import Options
`1 2 3 4 5` `6 7 8 9 10`	See "Importing Numeric Text Data" on page 6-37 or "Using the Import Wizard" on page 6-11.

`1; 2; 3; 4; 5`
`6; 7; 8; 9; 10`
`or`
`1, 2, 3, 4, 5`
`6, 7, 8, 9, 10`
	See "Importing Delimited ASCII Data Files" on page 6-38 or "Using the Import Wizard" on page 6-11.

`Grade1 Grade2 Grade3`
`91.5   89.2   77.3`
`88.0   67.8   91.0`
`67.3    78.1   92.5`
	See"Importing Data with Text Headers" on page 6-41 or "Using the Import Wizard" on page 6-11.

`Ann Type1 12.34 45 Yes` `Joe Type2 45.67 67 No`	See "Importing Mixed Alphabetic and Numeric Data" on page 6-39.

If you are familiar with MATLAB import functions, but are not sure when to use them, see the following table which compares the features of each function.

Table 6-2 ASCII Data Import Function Features

Function	Data Type	Delimiters	Notes
`csvread`	Numeric data	Commas only	Primarily used with spreadsheet data. See "Working with Spreadsheets" on page 6-49.
`dlmread`	Numeric data	Any character	Flexible and easy to use.
`fscanf`	Alphabetic and numeric; however, both types are returned in a single return variable	Any character	Part of low-level file I/O routines. Flexible, but requires more complex programming on your part.
`load`	Numeric data	Space, tab, comma, or semicolon character	Easy to use. File can include MATLAB-style comments.
`textscan`	Alphabetic and numeric data	Any character	Flexible, powerful, and easy to use. Use format string to specify conversions. File can include headers and any style of comments.

Importing Numeric Text Data

If your data file contains only numeric data, you can use many of the MATLAB import functions (listed in ASCII Data Import Function Features on page 6-37), depending on how the data is delimited. If the data is rectangular, that is, each row has the same number of elements, the simplest command to use is the `load` command. (You can also use the `load` function to import MAT-files, the MATLAB binary format for saving the workspace.)

For example, the file named `my_data.txt` contains two rows of numbers delimited by space characters:

```
1 2 3 4 5
6 7 8 9 10
```

When you use `load` as a command, it imports the data and creates a variable in the workspace with the same name as the filename, minus the file extension:

```
load my_data.txt;
whos
   Name          Size         Bytes   Class

   my_data       2x5           80    double array

my_data

my_data =
    1   2   3   4   5
    6   7   8   9   10
```

If you want to name the workspace variable something other than the filename, use the functional form of `load`. In the following example, the data from `my_data.txt` is loaded into the workspace variable `A`:

```
A = load('my_data.txt');
```

Importing Delimited ASCII Data Files

If your data file uses a character other than a space, tab, comma, or semicolon as a delimiter, or you would like to read a portion of your data, you have a choice of several import functions to use. (For a complete list, see ASCII Data Import Function Features on page 6-37.) The simplest function for numeric data is `dlmread`.

For example, consider a file named `ph.dat` whose contents are separated by a vertical line:

```
7.2|8.5|6.2|6.6
5.4|9.2|8.1|7.2
```

To read the entire contents of this file into an array named A, enter

```
A = dlmread('ph.dat', '|');
```

You specify the delimiter used in the data file as the second argument to dlmread. Note that, even though the last items in each row are not followed by a delimiter, dlmread can still process the file correctly. dlmread ignores space characters between data elements. So, the preceding dlmread command works even if the contents of ph.dat are

```
7.2|   8.5|       6.2|6.6
5.4|   9.2   |8.1|7.2
```

Importing Mixed Alphabetic and Numeric Data

If your data file contains a mix of alphabetic and numeric ASCII data, the simplest option for importing is the textscan function.

This example uses textscan to import the file mydata.dat as shown below:

```
Sally    Type1 12.34 45 Yes
Larry    Type2 34.56 54 Yes
Tommy    Type1 67.89 23 No
```

To read the entire contents of the file mydata.dat into the workspace, specify the name of the data file and the format string as arguments to textscan. In the format string, you include conversion specifiers that define how you want each data item to be interpreted. For example, specify %s for string data, %f for floating-point data, and so on. (For a complete list of format specifiers, see the textscan reference page.)

In this example, textscan reads the file mydata.dat, applying the format string to each line in the file until the end of the file:

```
fid = fopen('mydata.dat');
mydata = textscan(fid, '%s %s %f %d %s');

mydata{:}
ans =
    'Sally'
    'Larry'
    'Tommy'
```

```
ans =
    'Type1'
    'Type2'
    'Type1'

ans =
   12.3400
   34.5600
   67.8900

ans =
    45
    54
    23

ans =
    'Yes'
    'Yes'
    'No'

fclose(fid);
```

To remove the text 'Type' from each entry in the second column of the data, add the literal 'Type' to the beginning of the conversion specifier as shown below:

```
fid = fopen('mydata.dat');
mydata = textscan(fid, '%s Type%d %f %d %s');
fclose(fid);
```

Instead of returning a cell array of strings for mydata{2}, textscan returns a numeric array of type int32:

```
mydata{2}
ans =
    1
    2
    1
```

Importing Data with Text Headers

To import an ASCII data file that contains text headers, use the `textscan` function.

`textscan` accepts a set of predefined parameters that control various aspects of the conversion, including specifications for headers, delimiters, comments, and empty values. (For a complete list of these parameters, see the `textscan` reference page.)

Using the `headerlines` parameter, you can specify the number of lines at the head of the file that `textscan` should ignore.

For example, the file `grades.dat` contains formatted numeric data with a one-line text header:

```
Grade1 Grade2 Grade3
  78.8   55.9   45.9
  99.5   66.8   78.0
  89.5   77.0   56.7
```

To import this data, first open the file, and then use this `textscan` command to read the contents:

```
fid = fopen('grades.dat', 'r');
grades = textscan(fid, '%f %f %f', 'headerlines', 1);

grades{:}
ans =
   78.8000
   99.5000
   89.5000

ans =
   55.9000
   66.8000
   77.0000

ans =
   45.9000
   78.0000
```

```
   56.7000

fclose(fid);
```

Importing Mixed Alphabetic and Numeric Data with a Delimiter

If your data uses a character other than a space as a delimiter, use the `textscan` parameter `'delimiter'` to specify the delimiter. For example, if the file `grades.dat` used a semicolon as a delimiter, you would use this command:

```
grades =  textscan(fid, '%f %f %f', 'headerlines', 1, 'delimiter', ';')
```

Importing Large Data Sets

An efficient way to read files with large data sets is to read the file in segments and process the data as you go. This method requires significantly less memory than if you were to try reading in the entire file at once. Using the `textscan` function, you can read a specified amount of data from a file, and maintain a pointer to the location in the file where your last read operation ended and your next read is to begin.

This example opens a large data file and reads the file a segment at a time in a `for` loop. The code calls `textscan` to read a particular pattern of data (as specified by `format`) 10,000 times for each segment. Following each read, the subfunction `process_data` processes the data collected in cell array `segarray`:

```
format = '%s %n %s %8.2f %8.2f %8.2f %8.2f %u8';
file_id = fopen('largefile.dat', 'r');

for k = 1:segcount
   segarray = textscan(file_id, format, 10000);
   process_data(segarray);
end

fclose(file_id);
```

Importing from XML Documents

With the `xmlread` function, you can read from a given URL or file, generating a Document Object Model (DOM) node to represent the parsed document.

MATLAB also provides these other XML functions:

- `xmlwrite` — Serializes a Document Object Model node to a file
- `xslt` — Transforms an XML document using an XSLT engine

For more information, see the reference pages for these functions.

Exporting Text Data

In this section...

Overview

This section describes how to use MATLAB functions to export data in several common ASCII formats. For example, you can use these functions to export a MATLAB matrix as a text file where the rows and columns are represented as space-separated, numeric values. The function you use depends on the amount of data you want to export and its format.

If you are not sure which section describes your data, refer to the table that follows and find the sample in it that most nearly matches the data format you want to create. Then read the section referred to in the table.

If you are familiar with MATLAB export functions, but are not sure when to use them, see ASCII Data Export Function Features on page 6-45, which compares the features of each function.

Note If C or Fortran routines for writing data files in the form needed by other applications exist, create a MEX-file to write the data. For more information, see the MATLAB External Interfaces documentation.

Table 6-3 ASCII Data File Formats

Data Format Sample	MATLAB Export Function
`1 2 3 4 5 6` `7 8 9 10`	See "Exporting Delimited ASCII Data Files" on page 6-46 and "Using the diary Function to Export Data" on page 6-47 for information about these options.

`1; 2; 3; 4;`
`5; 6; 7; 8;`
`9; 10;`
	See "Exporting Delimited ASCII Data Files" on page 6-46. The example shows a semicolon-delimited file, but you can specify another character as the delimiter.

Table 6-4 ASCII Data Export Function Features

Function	Use With	Delimiters	Notes
`csvwrite`	Numeric data	Commas only	Primarily used with spreadsheet data. See "Working with Spreadsheets" on page 6-49.
`diary`	Numeric data or cell array	Spaces only	Use for small arrays. Requires editing of data file to remove extraneous text.
`dlmwrite`	Numeric data	Any character	Easy to use, flexible.
`fprintf`	Alphabetic and numeric data	Any character	Part of low-level file I/O routines. This function is the most flexible, but also the most difficult to use. You must use `fopen` to obtain a file identifier before writing the data and `fclose` to close the file after writing the data.
`save`	Numeric data	Tabs or spaces	Easy to use; output values are high precision.

Exporting Delimited ASCII Data Files

To export an array as a delimited ASCII data file, you can use either the `save` function, specifying the `-ASCII` qualifier, or the `dlmwrite` function. The `save` function is easy to use; however, the `dlmwrite` function provides more flexibility, allowing you to specify any character as a delimiter and to export subsets of an array by specifying a range of values.

Using the save Function

To export the array `A`,

```
A = [ 1 2 3 4 ; 5 6 7 8 ];
```

use the `save` function, as follows:

```
save my_data.out A -ASCII
```

If you view the created file in a text editor, it looks like this:

```
1.0000000e+000  2.0000000e+000  3.0000000e+000  4.0000000e+000
5.0000000e+000  6.0000000e+000  7.0000000e+000  8.0000000e+000
```

By default, `save` uses spaces as delimiters, but you can use tabs instead of spaces by specifying the `-tabs` option.

When you use `save` to write a character array to an ASCII file, it writes the ASCII equivalent of the characters to the file. If you write the character string `'hello'` to a file, `save` writes the values

```
104 101 108 108 111
```

Using the dlmwrite Function

To export an array in ASCII format and specify the delimiter used in the file, use the `dlmwrite` function.

For example, to export the array `A`,

```
A = [ 1 2 3 4 ; 5 6 7 8 ];
```

as an ASCII data file that uses semicolons as a delimiter, use this command:

```
dlmwrite('my_data.out',A, ';')
```

If you view the created file in a text editor, it looks like this:

```
1;2;3;4
5;6;7;8
```

Note that `dlmwrite` does not insert delimiters at the end of rows.

By default, if you do not specify a delimiter, `dlmwrite` uses a comma as a delimiter. You can specify a space (`' '`) as a delimiter or, if you specify empty quotes (`''`), no delimiter.

Using the diary Function to Export Data

To export small numeric arrays or cell arrays, you can use the `diary` function. `diary` creates a verbatim copy of your MATLAB session in a disk file (excluding graphics).

For example, if you have the array `A` in your workspace,

```
A = [ 1 2 3 4; 5 6 7 8 ];
```

execute these commands at the MATLAB prompt to export this array using `diary`:

1 Turn on the `diary` function. You can optionally name the output file `diary` creates.

```
diary my_data.out
```

2 Display the contents of the array you want to export. This example displays the array `A`. You could also display a cell array or other MATLAB class.

```
A =
     1     2     3     4
     5     6     7     8
```

3 Turn off the `diary` function.

```
diary off
```

diary creates the file my_data.out and records all the commands executed in the MATLAB session until it is turned off.

```
A =

     1     2     3     4
     5     6     7     8

diary off
```

4 Open the diary file my_data.out in a text editor and remove all the extraneous text.

Exporting to XML Documents

With the xmlwrite function, you can serialize a Document Object Model (DOM) node to an XML file.

MATLAB also provides these other XML functions:

- xmlread — Imports from a given URL or file to a Document Object Model node
- xslt — Transforms an XML document using an XSLT engine

For more information, see the reference pages for these functions.

Working with Spreadsheets

In this section…

Microsoft Excel Spreadsheets

This section includes:

Communicating with Excel Applications

Full functionality of the Excel import and export functions depends on the availability of the Excel COM Server. This server is part of the typical installation of Excel for Windows. If your system has Excel for Windows installed, then the import and export functions can read or write any file format recognized by your version of Excel.

However, if your system does not have Excel for Windows installed, or the COM server is not available:

- `xlswrite` writes your data to a comma-separated value (CSV) file.
- The import functions (`xlsread`, `importdata`, and the Import Wizard) only read XLS files compatible with Excel 97-2003.
- You can specify a worksheet to read in the Excel file with the `xlsread` function, but you cannot specify a range of data. See the `xlsread` reference page for additional information.

If you have Excel 2003 installed, but want to write to a 2007 format (such as XLSX, XLSB, or XLSM), you must install the Office 2007 Compatibility Pack.

Note Large files in XLSX format might load very slowly. For better import and export performance with Excel 2007 files, Microsoft recommends that you use the XLSB format.

Getting Information about a File

Use the `xlsfinfo` function to determine whether a file contains a readable Excel spreadsheet . For readable files, `xlsfinfo` returns the string `'Microsoft Excel Spreadsheet'`. Otherwise, it returns an empty string (`''`).

You also can use `xlsfinfo` to identify the names of the worksheets in the file, and to obtain the file format reported by Excel.

Example — Querying an XLS File. This example returns information about spreadsheet file `climate.xls`:

```
[type, sheets] = xlsfinfo('climate.xls')

type =
Microsoft Excel Spreadsheet
sheets =
    'Locations'    'Rainfall'    'Temperatures'
```

Exporting to a File

Use the `xlswrite` function to export a matrix to an Excel spreadsheet file. With `xlswrite`, you can export data from the workspace to any worksheet in the file, and to any location within that worksheet. By default, `xlswrite` writes your matrix data to the first worksheet in the file, starting at cell `A1`.

Example — Writing To an XLS File. This example writes a mix of text and numeric data to the file `climate.xls`. Call `xlswrite`, specifying a worksheet labeled `Temperatures`, and the region within the worksheet where you want to write the data. `xlswrite` writes the 4-by-2 matrix `d` to the rectangular region that starts at cell `E1` in its upper-left corner:

```
d = {'Time', 'Temp'; 12 98; 13 99; 14 97}
d =
    'Time'    'Temp'
    [  12]    [  98]
    [  13]    [  99]
    [  14]    [  97]

xlswrite('climate.xls', d, 'Temperatures', 'E1');
```

Adding a New Worksheet. If the target worksheet does not already exist in the file, `xlswrite` displays the following warning:

```
Warning: Added specified worksheet.
```

You can disable these warnings with this command:

```
warning off MATLAB:xlswrite:AddSheet
```

Importing from a File

There are several ways to import data from an Excel spreadsheet file into the MATLAB workspace:

- Use the Import Wizard. The simplest way to start the wizard is to select **Import Data** from the **File** menu. The Import Wizard automatically detects row and column headers, and gives you the option to create vectors from each row or column. However, you cannot select the worksheet or the range of data to import. For more information, see "Using the Import Wizard" on page 6-11.
- Call the `importdata` function. This function imports spreadsheet data into a structure. (For information on the fields in this structure, see Example 2.) As with the Import Wizard, you cannot select the worksheet or the range of data to import. For additional information, see the `importdata` reference page.
- Call the `xlsread` function. By default, `xlsread` imports all numeric data in the first worksheet of the file into a matrix. You can specify the worksheet and the range of data to import either by explicitly passing parameters to `xlsread`, or by using the syntax `xlsread(filename, -1)`. The `-1` option requests that `xlsread` open the file in an Excel window, so that you can

interactively select the worksheet and the range of data to import. For more information on options and syntax, see the xlsread reference page.

All import options support XLS and XLSX formats. The importdata and xlsread functions also import XLSB and XLSM formats, and xlsread imports HTML-based formats.

Example 1 — Reading from an XLS File with xlsread. Consider the file climate.xls created in the export example. To import only the numeric data into a matrix, use xlsread with a single return argument. xlsread ignores any leading row or column of text in the numeric result:

```
ndata = xlsread('climate.xls', 'Temperatures')
ndata =
    12    98
    13    99
    14    97
```

To import both numeric data and text data, specify two return values for xlsread:

```
[ndata, headertext] = xlsread('climate.xls', 'Temperatures')

headertext =
    'Time'    'Temp'

ndata =
    12    98
    13    99
    14    97
```

Example 2 — Reading from an XLS File with importdata. The importdata function reads data from an Excel file into a structure. Continuing the example above, where the data includes column headers, a call of the form

```
climate = importdata('climate.xls')      % with column headers
```

returns the nested structure array

```
climate =
```

```
           data: [1x1 struct]
       textdata: [1x1 struct]
     colheaders: [1x1 struct]
```

Structures created from Excel files with row headers include the field `rowheaders`, which also contains a 1-by-1 structure.

The structure named `data` contains one field for each worksheet with numeric data. The other structures contain one field for each worksheet with text cells or headers. In this case:

```
climate.data =
    Temperatures: [3x2 double]

climate.textdata =
    Temperatures: {'Time'  'Temp'}

climate.colheaders =
    Temperatures: {'Time'  'Temp'}
```

If the Excel file contains only numeric data (no row or column headers, and no inner cells with text), the output structure is simpler (a 1-by-1 structure, with one field for each worksheet with data).

For example, if the `Temperatures` worksheet in `climate_nc.xls` does not include column headers, the call

```
ndata = importdata('climate_nc.xls')      % only numeric data
```

returns

```
ndata =
    Temperatures: [3x2 double]
```

Converting Dates

Both Excel and MATLAB applications represent numeric dates as a number of serial days elapsed from a specific reference date. However, Excel and MATLAB use different reference dates. Therefore, you must convert any

numeric date that you import or export before you process it in your target application.

Note MATLAB functions import all formatted dates as strings. To import a numeric date, the date field in Excel must have a numeric format. See "Example 1 — Importing an Excel File with Numeric Dates" on page 6-54.

The following table lists the reference dates for MATLAB and Excel. For more information on the 1900 and 1904 date systems, see the Excel help.

Application	Reference Date
MATLAB	January 0, 0000
Excel for Windows	January 1, 1900
Excel for the Macintosh®	January 2, 1904

Example 1 — Importing an Excel File with Numeric Dates. Consider the file `weight_log.xls` with

```
Date        Weight
10/31/96    174.8
11/29/96    179.3
12/30/96    190.4
01/31/97    185.7
```

To import this file, first convert the dates within Excel to a numeric format. In Windows, the file now appears as

```
Date        Weight
35369       174.8
35398       175.3
35429       190.4
35461       185.7
```

Import the file:

```
wt = xlsread('weight_log.xls');
```

Convert the dates to the MATLAB reference date. If the file uses the 1900 date system (the default in Excel for Windows):

```
datecol = 1;
wt(:,datecol) = wt(:,datecol) + datenum('30-Dec-1899');
```

If the file uses the 1904 date system (the default in Excel for the Macintosh):

```
datecol = 1;
wt(:,datecol) = wt(:,datecol) + datenum('01-Jan-1904');
```

Example 2 — Exporting to an Excel File with Numeric Dates. To export the `wt` matrix from the previous example to an Excel file, first convert the dates according to the appropriate reference date. If you are exporting to Excel for Windows (and plan to use the default 1900 date system), convert as follows:

```
datecol = 1;
wt(:,datecol) = wt(:,datecol) - datenum('30-Dec-1899');
xlswrite('new_log.xls', wt);
```

If you are exporting to Excel for the Macintosh (and plan to use the default 1904 date system), convert as follows:

```
datecol = 1;
wt(:,datecol) = wt(:,datecol) - datenum('01-Jan-1904');
xlswrite('new_log.xls', wt);
```

Lotus 123 Spreadsheets

This section covers

- "Getting Information About the File" on page 6-56
- "Exporting to the File" on page 6-56
- "Importing from the File" on page 6-57

For more detailed information and examples, see the `wk1finfo`, `wk1write`, and `wk1read` reference pages.

Getting Information About the File

Use the `wk1finfo` function to determine if a file contains a Lotus WK1 spreadsheet.

Inputs to `wk1finfo` are:

- Name of the spreadsheet file

Outputs from `wk1finfo` are:

- String `'WK1'` if the file is a Lotus spreadsheet readable with the `wk1read` function. Otherwise, it contains an empty string (`''`).
- String `'Lotus 123 Spreadsheet'`

Example — Querying a WK1 File. This example returns information about spreadsheet file `matA.wk1`:

```
[extens, type] = wk1finfo('matA.wk1')

extens =
   WK1
type =
   Lotus 123 Spreadsheet
```

Exporting to the File

Use the `wk1write` function to export a matrix to a Lotus spreadsheet file. You have the choice of positioning the matrix starting at the first row and column of the spreadsheet, or at any other location in the file.

To export to a specific location in the file, use the second syntax, indicating a zero-based starting row and column.

Inputs to `wk1write` are:

- Name of the spreadsheet file
- Matrix to be exported
- Location in the file in which to write the data

Example — Writing to a WK1 File. This example exports an 8-by-8 matrix to spreadsheet file `matA.wk1`:

```
A = [1:8; 11:18; 21:28; 31:38; 41:48; 51:58; 61:68; 71:78];
A =
     1     2     3     4     5     6     7     8
    11    12    13    14    15    16    17    18
    21    22    23    24    25    26    27    28
    31    32    33    34    35    36    37    38
    41    42    43    44    45    46    47    48
    51    52    53    54    55    56    57    58
    61    62    63    64    65    66    67    68
    71    72    73    74    75    76    77    78

wk1write('matA.wk1', A);
```

Importing from the File

To import data from the spreadsheet into the MATLAB workspace, use `wk1read`. There are three ways to call `wk1read`. The first two shown here are similar to `wk1write`. The third enables you to select a range of values from the spreadsheet. You can specify the `range` argument with a one-based vector, spreadsheet notation (e.g., `'A1..B7'`), or using a named range (e.g., `'Sales'`).

Inputs to `wk1read` are:

- Name of the spreadsheet file
- Spreadsheet location from which to read the data
- Range of cells from which to read the data

Outputs from `wk1read` are:

- Requested data from the spreadsheet

Example — Reading from a WK1 File. Read in a limited block of the spreadsheet data by specifying the upper-left row and column of the block using zero-based indexing:

```
M = wk1read('matA.wk1', 3, 2)
M =
    33    34    35    36    37    38
    43    44    45    46    47    48
    53    54    55    56    57    58
    63    64    65    66    67    68
    73    74    75    76    77    78
```

Working with Graphics Files

In this section...

Getting Information About Graphics Files

If you have a file in a standard graphics format, use the `imfinfo` function to get information about its contents. The `imfinfo` function returns a structure containing information about the file. The fields in the structure vary with the file format, but `imfinfo` always returns some basic information including filename, last modification date, file size, and format.

This example returns information about a file in Joint Photographic Experts Group (JPEG) format:

```
info = imfinfo('ngc6543a.jpg')

info =

           Filename: [1x57 char]
        FileModDate: '01-Oct-1996 16:19:44'
           FileSize: 27387
             Format: 'jpg'
      FormatVersion: ''
              Width: 600
             Height: 650
           BitDepth: 24
          ColorType: 'truecolor'
    FormatSignature: ''
    NumberOfSamples: 3
       CodingMethod: 'Huffman'
      CodingProcess: 'Sequential'
            Comment: {[1x69 char]}
```

Importing Graphics Data

To import data into the MATLAB workspace from a graphics file, use the `imread` function. Using this function, you can import data from files in many standard file formats, including the Tagged Image File Format (TIFF), Graphics Interchange Format (GIF), Joint Photographic Experts Group (JPEG), and Portable Network Graphics (PNG) formats. For a complete list of supported formats, see the `imread` reference page.

This example reads the image data stored in a file in JPEG format into the MATLAB workspace as the array `I`:

```
I = imread('ngc6543a.jpg');
```

`imread` represents the image in the workspace as a multidimensional array of class `uint8`. The dimensions of the array depend on the format of the data. For example, `imread` uses three dimensions to represent RGB color images:

```
whos I
  Name      Size                         Bytes  Class

  I       650x600x3                    1170000  uint8 array

Grand total is 1170000 elements using 1170000 bytes
```

Exporting Graphics Data

To export data from the MATLAB workspace using one of the standard graphics file formats, use the `imwrite` function. Using this function, you can export data in formats such as the Tagged Image File Format (TIFF), Joint Photographic Experts Group (JPEG), and Portable Network Graphics (PNG). For a complete list of supported formats, see the `imwrite` reference page.

The following example writes a multidimensional array of `uint8` data `I` from the MATLAB workspace into a file in TIFF format. The class of the output image written to the file depends on the format specified. For most formats, if the input array is of class `uint8`, `imwrite` outputs the data as 8-bit values. See the `imwrite` reference page for details.

```
whos I
  Name      Size                         Bytes  Class
```

```
  I       650x600x3                     1170000  uint8 array

Grand total is 1170000 elements using 1170000 bytes
imwrite(I, 'my_graphics_file.tif','tif');
```

Note `imwrite` supports different syntaxes for several of the standard formats. For example, with TIFF file format, you can specify the type of compression MATLAB uses to store the image. See the `imwrite` reference page for details.

Working with Audio and Video Data

In this section...

Getting Information About Audio/Video Files

The MATLAB software includes several functions that you can use to get information about files that contain audio data, video data, or both. Some work only with specific file formats. One function, the `mmfileinfo` function, can retrieve information about many file formats.

Format-Specific Functions

MATLAB includes several functions that return information about files that contain audio and video data in specific formats.

- `aufinfo` — Returns a text description of the contents of a sound (AU) file
- `aviinfo` — Returns a structure containing information about the contents of an Audio/Video Interleaved (AVI) file
- `wavfinfo` — Returns a text description of the contents of a sound (WAV) file

Using the General Multimedia Information Function

MATLAB also includes a general-purpose, audio/video file information function named `mmfileinfo`. The `mmfileinfo` function returns information about both the audio data in a file and the video data in the file, if present.

Note You can use `mmfileinfo` only on Microsoft Windows systems.

Importing Audio/Video Data

MATLAB includes several functions that you can use to bring audio or video data into the MATLAB workspace. Some of these functions read audio or

video data from files. Another way to import audio data into the MATLAB workspace is to record it using an audio input device, such as a microphone. The following sections describe

- “Reading Audio and Video Data from a File” on page 6-63
- “Recording Audio Data” on page 6-63

Reading Audio and Video Data from a File

MATLAB includes several functions for reading audio or video data from a file. These files are format-specific.

- `auread` — Returns sound data from a sound (AU) file
- `aviread` — Returns AVI data as a MATLAB movie
- `mmreader` — Returns AVI, MPG, or WMV video data
- `wavread` — Returns sound data from a sound (WAV) file

Note You can use `mmreader` only on Microsoft Windows systems.

Recording Audio Data

To bring sound data into the MATLAB workspace by recording it from an audio input device, use the audio recorder object. This object represents the connection between MATLAB and an audio input device, such as a microphone, that is connected to your system. You use the `audiorecorder` function to create this object, and then you use methods and properties of the object to record the audio data.

On PCs running the Windows operating system, you also can use the `wavrecord` function to bring live audio data in WAV format into the MATLAB workspace.

Once you import audio data, MATLAB supports several ways to listen to the data. You can use an audio player object to play the audio data. Use the `audioplayer` function to create an audio player object.

You also can use the `sound` or `soundsc` function.

On PCs running the Windows operating system, you can use the `wavplay` function to listen to `.wav` files.

Exporting Audio/Video Data

MATLAB includes several functions that you can use to export audio or video data from the MATLAB workspace. These functions write audio data to a file using specific file formats. The following sections describe

- "Exporting Audio Data" on page 6-64
- "Exporting Video Data in AVI Format" on page 6-64

For an example of writing video data to a file, see "Example: Creating an AVI file" on page 6-65.

Exporting Audio Data

In MATLAB, audio data is simply numeric data that you can export using standard MATLAB data export functions, such as `save`.

MATLAB also includes several functions that write audio data to files in specific file formats:

- `auwrite` — Exports sound data in AU file format
- `wavwrite` — Exports sound data in WAV file format

Exporting Video Data in AVI Format

You can export MATLAB video data as an Audio/Video Interleaved (AVI) file. To do this, you use the `avifile` function to create an `avifile` object. When you have the object, you can use AVI file object methods and properties to control various aspects of the data export process.

For example, in MATLAB, you can save a sequence of graphs as a movie that you then can play back using the `movie` function. You can export a MATLAB movie by saving it in MAT-file format, like any other MATLAB workspace variable. (However, only people running MATLAB can view your movie.) For

more information about MATLAB movies, see the Animation section in the MATLAB Graphics documentation.

To export a sequence of MATLAB graphs in a format that does not require MATLAB for viewing, save the figures in Audio/Video Interleaved (AVI) format. AVI is a file format that allows animation and video clips to be played on a PC running Windows or The Open Group UNIX operating systems.

Note To convert an existing MATLAB movie into an AVI file, use the `movie2avi` function.

Example: Creating an AVI file

To export a sequence of MATLAB graphs as an AVI format movie, perform these steps:

1. Create an AVI file object, using the `avifile` function.

   ```
   aviobj = avifile('mymovie.avi','fps',5);
   ```

   AVI file objects support properties that let you control various characteristics of the AVI movie, such as colormap, compression, and quality. (For a complete list, see the `avifile` reference page.) `avifile` uses default values for all properties, unless you specify a value. The example sets the value of the frames per second (`fps`) property.

2. Capture the sequence of graphs and put them into the AVI file, using the `addframe` function.

   ```
   for k=1:25
       h = plot(fft(eye(k+16)));
       axis equal;
       frame = getframe(gca);
       aviobj = addframe(aviobj,frame);
   end
   ```

   The example uses a `for` loop to capture the series of graphs to be included in the movie. You typically use `addframe` to capture a sequence of graphs for AVI movies. However, because this particular MATLAB animation uses

`XOR` graphics, you must call `getframe` to capture the graphs, and then call `addframe` to add the captured frame to the movie.

3 Close the AVI file, using the `close` function.

```
aviobj = close(aviobj);
```

Using Low-Level File I/O Functions

In this section...

Overview

The MATLAB product includes a set of low-level file I/O functions that are based on the I/O functions of the American National Standards Institute's ANSI® Standard C Library. If you know C, you are probably familiar with these routines.

To read or write data, perform these steps:

1 Open the file, using `fopen`. `fopen` returns a file identifier that you use with all the other low-level file I/O routines.

2 Operate on the file.

   a Read binary data, using `fread`.

   b Write binary data, using `fwrite`.

   c Read text strings from a file line-by-line, using `fgets` or `fgetl`.

   d Read formatted ASCII data, using `fscanf`.

   e Write formatted ASCII data, using `fprintf`.

3 Close the file, using `fclose`.

This section also describes how these functions affect the current position in the file where read or write operations happen and how you can change the position in the file.

Note While the MATLAB file I/O commands are modeled on the C language I/O routines, in some ways their behavior is different. For example, the `fread` function is *vectorized*; that is, it continues reading until it encounters a text string or the end of file. These sections, and the MATLAB reference pages for these functions, highlight any differences in behavior.

Opening Files

Before reading or writing a text or binary file, you must open it with the `fopen` command.

```
fid = fopen('filename','permission')
```

Specifying the Permission String

The `permission` string specifies the kind of access to the file you require. Possible `permission` strings include:

- `r` for reading only
- `w` for writing only
- `a` for appending only
- `r+` for both reading and writing

Note Systems such as Microsoft Windows operating systems that distinguish between text and binary files might require additional characters in the permission string, such as `'rb'` to open a binary file for reading.

Using the Returned File Identifier (fid)

If successful, `fopen` returns a nonnegative integer, called a *file identifier* (`fid`). You pass this value as an argument to the other I/O functions to access

the open file. For example, this `fopen` statement opens the data file named `penny.dat` for reading:

```
fid = fopen('penny.dat','r')
```

If `fopen` fails, for example if you try to open a file that does not exist, `fopen` does the following:

- Assigns `-1` to the file identifier.
- Assigns an error message to an optional second output argument. Note that the error messages are system dependent and are not provided for all errors on all systems. The function `ferror` also can provide information about errors.

Test the file identifier each time you open a file in your code. For example, this code loops until a readable filename is entered:

```
fid=0;
while fid < 1
   filename=input('Open file: ', 's');
   [fid,message] = fopen(filename, 'r');
   if fid == -1
     disp(message)
   end
end
```

When you run this code, if you specify a file that doesn't exist, such as `nofile.mat`, at the `Open file:` prompt, the results are as follows:

```
Open file: nofile.mat
Sorry. No help in figuring out the problem . . .
```

If you specify a file that does exist, such as `goodfile.mat`, the code example returns the file identifier, `fid`, and exits the loop:

```
Open file: goodfile.mat
```

Opening Temporary Files and Directories

The tempdir and tempname functions assist in locating temporary data on your system.

Function	Purpose
tempdir	Get temporary directory name.
tempname	Get temporary filename.

Use these functions to create temporary files. Some systems delete temporary files every time you reboot the system. On other systems, designating a file as temporary can mean only that the file is not backed up.

The tempdir function returns the name of the directory or folder that has been designated to hold temporary files on your system. For example, issuing tempdir on The Open Group UNIX systems returns the /tmp directory.

MATLAB also provides a tempname function that returns a filename in the temporary directory. The returned filename is a suitable destination for temporary data. For example, if you need to store some data in a temporary file, then you might issue the following command first:

```
fid = fopen(tempname, 'w');
```

Note The filename that tempname generates is not guaranteed to be unique; however, it is likely to be so.

Reading Binary Data

The fread function reads all or part of a binary file (as specified by a file identifier) and stores it in a matrix. In its simplest form, it reads an entire file and interprets each byte of input as the next element of the matrix. For example, the following code reads the data from a file named nickel.dat into matrix A:

```
fid = fopen('nickel.dat','r');
A = fread(fid);
```

To echo the data to the screen after reading it, use `char` to display the contents of `A` as characters, transposing the data so it is displayed horizontally:

```
disp(char(A'))
```

The `char` function causes MATLAB to interpret the contents of `A` as characters instead of as numbers. Transposing `A` displays it in its more natural horizontal format.

Controlling the Number of Values Read

`fread` accepts an optional second argument that controls the number of values read (if unspecified, the default is the entire file). For example, this statement reads the first 100 data values of the file specified by `fid` into the column vector `A`.

```
A = fread(fid,100);
```

Replacing the number `100` with the matrix dimensions `[10 10]` reads the same 100 elements into a 10-by-10 array.

Controlling the Data Type of Each Value

An optional third argument to `fread` controls the class of the input. The class argument controls both the number of bits read for each value and the interpretation of those bits as character, integer, or floating-point values. MATLAB supports a wide range of precisions, which you can specify with MATLAB specific strings or their C or Fortran equivalents.

Some common precisions include:

- `'char'` and `'uchar'` for signed and unsigned characters (usually 8 bits)
- `'short'` and `'long'` for short and long integers (usually 16 and 32 bits, respectively)
- `'float'` and `'double'` for single- and double-precision floating-point values (usually 32 and 64 bits, respectively)

Note The meaning of a given precision can vary across different hardware platforms. For example, a `'uchar'` is not always 8 bits. `fread` also provides a number of more specific precisions, such as `'int8'` and `'float32'`. If in doubt, use precisions that are not platform dependent. For a complete list of precisions, see `fread`.

For example, if `fid` refers to an open file containing single-precision floating-point values, then the following command reads the next 10 floating-point values into a column vector `A`:

```
A = fread(fid,10,'float');
```

Writing Binary Data

The `fwrite` function writes the elements of a matrix to a file in a specified numeric precision, returning the number of values written. For instance, these lines create a 100-byte binary file containing the 25 elements of the 5-by-5 magic square, each stored as 4-byte integers:

```
fwriteid = fopen('magic5.bin','w');
count = fwrite(fwriteid,magic(5),'int32');
status = fclose(fwriteid);
```

In this case, `fwrite` sets the `count` variable to `25` unless an error occurs, in which case the value is less.

Controlling Position in a File

When you open a file with `fopen`, MATLAB maintains a file position indicator that specifies a particular location within a file. MATLAB uses the file position indicator to determine where in the file the next read or write operation will begin. The following sections describe how to:

- Determine whether the file position indicator is at the end of the file
- Move to a specific location in the file
- Retrieve the current location of the file position indicator
- Reset the file position indicator to the beginning of the file

Setting and Querying the File Position

The `fseek` and `ftell` functions enable you to set and query the position in the file at which the next input or output operation takes place:

- The `fseek` function repositions the file position indicator, letting you skip over data or back up to an earlier part of the file.
- The `ftell` function gives the offset in bytes of the file position indicator for a specified file.

The syntax for `fseek` is

```
status = fseek(fid,offset,origin)
```

`fid` is the file identifier for the file. `offset` is a positive or negative offset value, specified in bytes. `origin` is one of the following strings that specify the location in the file from which to calculate the position.

`'bof'`	Beginning of file
`'cof'`	Current position in file
`'eof'`	End of file

Example of Using fseek And ftell

To see how `fseek` and `ftell` work, consider this short M-file:

```
A = 1:5;
fid = fopen('five.bin','w');
fwrite(fid, A,'short');
status = fclose(fid);
```

This code writes out the numbers 1 through 5 to a binary file named `five.bin`. The call to `fwrite` specifies that each numerical element be stored as a `short`. Consequently, each number uses 2 storage bytes.

Now reopen `five.bin` for reading:

```
fid = fopen('five.bin','r');
```

This call to `fseek` moves the file position indicator forward 6 bytes from the beginning of the file:

```
status = fseek(fid,6,'bof');
```

File Position	bof	1	2	3	4	5	6	7	8	9	10	eof
File Contents		0	1	0	2	0	3	0	4	0	5	
File Position Indicator								↑				

This call to `fread` reads whatever is at file positions 7 and 8 and stores it in variable `four`:

```
four = fread(fid,1,'short');
```

The act of reading advances the file position indicator. To determine the current file position indicator, call `ftell`:

```
position = ftell(fid)

position =

     8
```

File Position	bof	1	2	3	4	5	6	7	8	9	10	eof
File Contents		0	1	0	2	0	3	0	4	0	5	
File Position Indicator										↑		

This call to `fseek` moves the file position indicator back 4 bytes:

```
status = fseek(fid,-4,'cof');
```

File Position	bof	1	2	3	4	5	6	7	8	9	10	eof
File Contents		0	1	0	2	0	3	0	4	0	5	
File Position Indicator						↑						

Calling `fread` again reads in the next value (3):

```
three = fread(fid,1,'short');
```

Reading Strings Line by Line from Text Files

MATLAB provides two functions, `fgetl` and `fgets`, that read lines from formatted text files and store them in string vectors. The two functions are almost identical; the only difference is that `fgets` copies the newline character to the string vector, but `fgetl` does not.

The following M-file function demonstrates a possible use of `fgetl`. This function uses `fgetl` to read an entire file one line at a time. For each line, the function determines whether an input literal string (`literal`) appears in the line.

If it does, the function prints the entire line preceded by the number of times the literal string appears on the line.

```
function y = litcount(filename, literal)
% Search for number of string matches per line.

fid = fopen(filename, 'rt');
y = 0;
while feof(fid) == 0
   tline = fgetl(fid);
   matches = findstr(tline, literal);
   num = length(matches);
   if num > 0
      y = y + num;
      fprintf(1,'%d:%s\n',num,tline);
   end
end
fclose(fid);
```

For example, consider the following input data file called `badpoem`:

```
Oranges and lemons,
Pineapples and tea.
Orangutans and monkeys,
Dragonflys or fleas.
```

To find out how many times the string `'an'` appears in this file, use `litcount`:

```
litcount('badpoem','an')
2: Oranges and lemons,
```

```
1: Pineapples and tea.
3: Orangutans and monkeys,
```

Reading Formatted ASCII Data

The MATLAB `fscanf` function is like the `fscanf` function in standard C. Both functions operate in a similar manner, reading data from a file and assigning it to one or more variables. Both functions use the same set of conversion specifiers to control the interpretation of the input data.

The conversion specifiers for `fscanf` begin with a `%` character; common conversion specifiers include the following.

Conversion Specifier	Description
`%s`	Match a string.
`%d`	Match an integer in base 10 format.
`%g`	Match a double-precision floating-point value.

You also can specify that `fscanf` skip a value by specifying an asterisk in a conversion specifier. For example, `%*f` means skip the floating-point value in the input data; `%*d` means skip the integer value in the input data.

Differences Between the MATLAB fscanf and the C fscanf

Despite all the similarities between the MATLAB and C versions of `fscanf`, there are some significant differences. For example, consider a file named `moon.dat` for which the contents are as follows:

```
3.654234533
2.71343142314
5.34134135678
```

The following code reads all three elements of this file into a matrix named `MyData`:

```
fid = fopen('moon.dat','r');
MyData = fscanf(fid,'%g');
status = fclose(fid);
```

Notice that this code does not use any loops. Instead, the `fscanf` function continues to read in text as long as the input format is compatible with the format specifier.

An optional size argument controls the number of matrix elements read. For example, if `fid` refers to an open file containing strings of integers, then this line reads 100 integer values into the column vector `A`:

```
A = fscanf(fid,'%5d',100);
```

This line reads 100 integer values into the 10-by-10 matrix `A`:

```
A = fscanf(fid,'%5d',[10 10]);
```

A related function, `sscanf`, takes its input from a string instead of a file. For example, this line returns a column vector containing `2` and its square root:

```
root2 = num2str([2, sqrt(2)]);
rootvalues = sscanf(root2,'%f');
```

Writing Formatted Text Files

The `fprintf` function converts data to character strings and outputs them to the screen or a file. A format control string containing conversion specifiers and any optional text specify the output format. The conversion specifiers control the output of array elements; `fprintf` copies text directly.

Common conversion specifiers include

Conversion Specifier	Description
`%e`	Exponential notation
`%f`	Fixed-point notation
`%g`	Automatically select the shorter of `%e` and `%f`

Optional fields in the format specifier control the minimum field width and precision. For example, this code creates a text file containing a short table of the exponential function:

```
x = 0:0.1:1;
y = [x; exp(x)];
```

The code below writes x and y into a newly created file named exptable.txt:

```
fid = fopen('exptable.txt','w');
fprintf(fid,'Exponential Function\n\n');
fprintf(fid,'%6.2f  %12.8f\n',y);
status = fclose(fid);
```

The first call to fprintf outputs a title, followed by two carriage returns. The second call to fprintf outputs the table of numbers. The format control string specifies the format for each line of the table:

- A fixed-point value of 6 characters with two decimal places
- Two spaces
- A fixed-point value of 12 characters with eight decimal places

fprintf converts the elements of array y in column order. The function uses the format string repeatedly until it converts all the array elements.

Now use fscanf to read the exponential data file:

```
fid = fopen('exptable.txt','r');
title = fgetl(fid);
[table,count] = fscanf(fid,'%f %f',[2 11]);
table = table';
status = fclose(fid);
```

The second line reads the file title. The third line reads the table of values, two floating-point values on each line, until it reaches end of file. count returns the number of values matched.

A function related to fprintf, sprintf, outputs its results to a string instead of a file or the screen. For example:

```
root2 = sprintf('The square root of %f is %10.8e.\n',2,sqrt(2));
```

Closing a File

When you finish reading or writing, use fclose to close the file. For example, this line closes the file associated with file identifier fid:

```
status = fclose(fid);
```

This line closes all open files:

```
status = fclose('all');
```

Both forms return `0` if the file or files were successfully closed or `-1` if the attempt was unsuccessful.

MATLAB automatically closes all open files when you exit from MATLAB. It is still good practice, however, to close a file explicitly with `fclose` when you are finished using it. Not doing so can unnecessarily drain system resources.

Note Closing a file does not clear the file identifier variable `fid`. Therefore, subsequent attempts to access a file through this file identifier variable will not work.

Accessing Files with Memory-Mapping

In this section...

Overview of Memory-Mapping

Memory-mapping is a mechanism that maps a portion of a file, or an entire file, on disk to a range of addresses within an application's address space. The application can then access files on disk in the same way it accesses dynamic memory. This makes file reads and writes faster in comparison with using functions such as `fread` and `fwrite`.

Another advantage of using memory-mapping in your MATLAB application is that it enables you to access file data using standard MATLAB indexing operations. Once you have mapped a file to memory, you can read the contents of that file using the same type of MATLAB statements used to read variables from the MATLAB workspace. The contents of the mapped file appear as if they were an array in the currently active workspace. You simply index into this array to read or write the desired data from the file.

This section describes the benefits and limitations of memory-mapping in MATLAB. The last part of this section gives details on which types of applications derive the greatest advantage from using memory-mapping:

- "Benefits of Memory-Mapping" on page 6-81
- "Limitations of MATLAB Memory-Mapping" on page 6-82
- "When to Use Memory-Mapping" on page 6-83

Benefits of Memory-Mapping

The principal benefits of memory-mapping are efficiency, faster file access, the ability to share memory between applications, and more efficient coding.

Faster File Access. Accessing files via memory map is faster than using I/O functions such as `fread` and `fwrite`. Data are read and written using the virtual memory capabilities that are built in to the operating system rather than having to allocate, copy into, and then deallocate data buffers owned by the process.

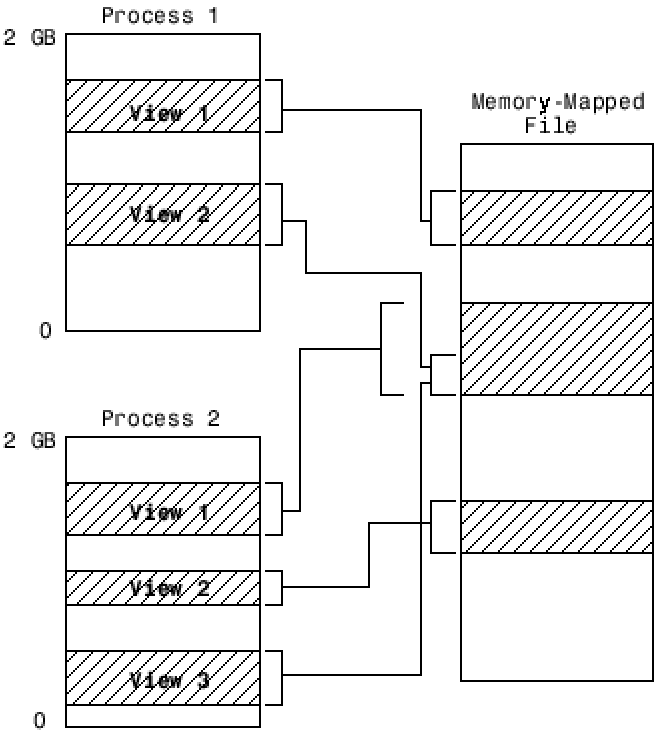
MATLAB does not access data from the disk when the map is first constructed. It only reads or writes the file on disk when a specified part of the memory map is accessed, and then it only reads that specific part. This provides faster random access to the mapped data.

Efficiency. Mapping a file into memory allows access to data in the file as if that data had been read into an array in the application's address space. Initially, MATLAB only allocates address space for the array; it does not actually read data from the file until you access the mapped region. As a result, memory-mapped files provide a mechanism by which applications can access data segments in an extremely large file without having to read the entire file into memory first.

Efficient Coding Style. Memory-mapping eliminates the need for explicit calls to the `fread` and `fwrite` functions. In MATLAB, if `x` is a memory-mapped variable, and `y` is the data to be written to a file, then writing to the file is as simple as

```
x.Data = y;
```

Sharing Memory Between Applications. Memory-mapped files also provide a mechanism for sharing data between applications, as shown in the figure below. This is achieved by having each application map sections of the same file. You can use this feature to transfer large data sets between MATLAB and other applications.



Also, within a single application, you can map the same segment of a file more than once.

Limitations of MATLAB Memory-Mapping

MATLAB restricts the size of a memory map to 2 gigabytes, and on some platforms, requires that you set up your memory-mapping so that all data access is aligned properly. See the following section, “Maximum Size of a Memory Map”, for more information.

Maximum Size of a Memory Map. Due to limits set by the operating system and MATLAB, the maximum amount of data you can map with a single instance of a memory map is 2 gigabytes on 32-bit systems, and 256 terabytes on 64-bit systems. If you need to map more than this limit, you can either create separate maps for different regions of the file, or you can move the window of one map to different locations in the file.

Aligned Access on Sol64. The Sol64 platform only supports aligned data access. This means that numeric values of type `double` that are to be read from a memory-mapped file must start at some multiple of 8 bytes from the start of the file. (Note that this is from the start of the *file*, and not the start of the mapped region.) Furthermore, numeric values of type `single` and also 32-bit integers must start at multiples of 4 bytes, and 16-bit integers at 2-byte multiples.

If you attempt to map a file on Sol64, which does not take into account these alignment considerations, MATLAB generates an error.

Byte Ordering

Memory-mapping works only with data that have the same byte ordering scheme as the native byte ordering of your operating system. For example, because both Linus Torvald's Linux and Microsoft Windows systems use little-endian byte ordering, data created on a Linux system can be read on Windows systems. You can use the `computer` function to determine the native byte ordering of your current system.

When to Use Memory-Mapping

Just how much advantage you get from mapping a file to memory depends mostly on the size and format of the file, the way in which data in the file is used, and the computer platform you are using.

When Memory-Mapping Is Most Useful. Memory-mapping works best with binary files, and in the following scenarios:

- For large files that you want to access randomly one or more times
- For small files that you want to read into memory once and access frequently

- For data that you want to share between applications
- When you want to work with data in a file as if it were a MATLAB array

When the Advantage Is Less Significant. The following types of files do not fully use the benefits of memory-mapping:

- Formatted binary files like HDF or TIFF that require customized readers are not good for memory-mapping. Describing the data contained in these files can be a very complex task. Also, you cannot access data directly from the mapped segment, but must instead create arrays to hold the data.
- Text or ASCII files require that you convert the text in the mapped region to an appropriate type for the data to be meaningful. This takes up additional address space.
- Files that are larger than several hundred megabytes in size consume a significant amount of the virtual address space needed by MATLAB to process your program. Mapping files of this size may result in MATLAB reporting out-of-memory errors more often. This is more likely if MATLAB has been running for some time, or if the memory used by MATLAB becomes fragmented.

The memmapfile Class

MATLAB implements memory-mapping using an object-oriented class called `memmapfile`. The `memmapfile` class has the properties and methods you need to map to a file, read and write the file via the map, and remove the map from memory when you are done.

Properties of the memmapfile Class

There are six properties defined for the `memmapfile` class. These are shown in the table below. These properties control which file is being mapped, where in the file the mapping is to begin and end, how the contents of the file are to be formatted, and whether or not the file is writable. One property of the file contains the file data itself.

Property	Description	Data Type	Default
`Data`	Contains the data read from the file or to be written to the file. (See "Reading a Mapped File" on page 6-100 and "Writing to a Mapped File" on page 6-105)	Any of the numeric types	None

Property	Description	Data Type	Default
`Filename`	Path and name of the file to map into memory. (See "Selecting the File to Map" on page 6-89)	`char` array	None
`Format`	Format of the contents of the mapped region, including class, array shape, and variable or field name by which to access the data. (See "Identifying the Contents of the Mapped Region" on page 6-91)	`char` array or N-by-3 cell array	`uint8`
`Offset`	Number of bytes from the start of the file to the start of the mapped region. This number is zero-based. That is, offset 0 represents the start of the file. Must be a nonnegative integer value. (See "Setting the Start of the Mapped Region" on page 6-90)	`double`	`0`
`Repeat`	Number of times to apply the specified format to the mapped region of the file. Must be a positive integer value or `Inf`. (See "Repeating a Format Scheme" on page 6-98)	`double`	`Inf`
`Writable`	Type of access allowed to the mapped region. Must be logical 1 or logical 0. (See "Setting the Type of Access" on page 6-99)	`logical`	`false`

You can set the values for any property except for `Data` at the time you call the `memmapfile` constructor, or at any time after that while the map is still valid. Any properties that are not explicitly set when you construct the object are given their default values as shown in the table above. For information on calling the constructor, see "Constructing a memmapfile Object" on page 6-86.

Once a `memmapfile` object has been constructed, you can change the value of any of its properties. Use the `objname.property` syntax in assigning the new value. For example, to set a new `Offset` value for memory map object `m`, type

```
m.Offset = 2048;
```

Note Property names are not case sensitive. For example, MATLAB considers `m.offset` to be the same as `m.Offset`.

To display the value of all properties of a memmapfile object, simply type the object name. For a memmapfile object m, typing the variable name m displays the following. Note that this example requires the file records.dat which you will create at the beginning of the next section.

```
m =
    Filename: 'records.dat'
    Writable: true
      Offset: 1024
      Format: 'uint32'
      Repeat: Inf
        Data: 4778x1 uint32 array
```

To display the value of any individual property, for example the Writable property of object m, type

```
m.Writable
ans =
   true
```

Constructing a memmapfile Object

The first step in mapping to any file is to construct an instance of the memmapfile class using the class constructor function. You can have MATLAB assign default values to each of the new object's properties, or you can specify property values yourself in the call to the memmapfile constructor.

For information on how to set these values, see these sections:

- "Constructing the Object with Default Property Values" on page 6-87
- "Changing Property Values" on page 6-88
- "Selecting the File to Map" on page 6-89
- "Setting the Start of the Mapped Region" on page 6-90
- "Identifying the Contents of the Mapped Region" on page 6-91
- "Mapping of the Example File" on page 6-96
- "Repeating a Format Scheme" on page 6-98
- "Setting the Type of Access" on page 6-99

All the examples in this section use a file named `records.dat` that contains a 5000-by-1 matrix of double-precision floating point numbers. Use the following code to generate this file before going on to the next sections of this documentation.

First, save this function in your current working directory:

```
function gendatafile(filename, count)
dmax32 = double(intmax('uint32'));
rand('state', 0)

fid = fopen(filename,'w');
fwrite(fid, rand(count,1)*dmax32, 'double');
fclose(fid);
```

Now execute the `gendatafile` function to generate the `records.dat` file that is referenced in this section. You can use this function at any time to regenerate the file:

```
gendatafile('records.dat', 5000);
```

Constructing the Object with Default Property Values

The simplest and most general way to call the constructor is with one input argument that specifies the name of the file you want to map. All other properties are optional and are given their default values. Use the syntax shown here:

```
objname = memmapfile(filename)
```

To construct a map for the file `records.dat` that resides in your current working directory, type the following:

```
m = memmapfile('records.dat')
m =
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: false
      Offset: 0
      Format: 'uint8'
      Repeat: Inf
        Data: 40000x1 uint8 array
```

MATLAB constructs an instance of the memmapfile class, assigns it to the variable m, and maps the entire records.dat file to memory, setting all object properties to their default values. In this example, the command maps the entire file as a sequence of unsigned 8-bit integers and gives the caller read-only access to its contents.

Changing Property Values

You can make the memory map more specific to your needs by including more information when calling the constructor. In addition to the filename argument, there are four other parameters that you can pass to the constructor. Each of these parameters represents a property of the object, and each requires an accompanying value to be passed, as well:

```
objname = memmapfile(filename, prop1, value1, prop2, value2, ...)
```

For example, to construct a map using nondefault values for the Offset, Format, and Writable properties, type the following, enclosing all property names and string parameter values in quotes:

```
m = memmapfile('records.dat', ...
      'Offset', 1024,         ...
      'Format', 'double',     ...
      'Writable', true);
```

Type the object name to see the current settings for all properties:

```
m

m =
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: true
      Offset: 1024
      Format: 'double'
      Repeat: Inf
        Data: 4872x1 double array
```

You can also change the value of any property after the object has been constructed. Use the syntax:

```
objname.property = newvalue;
```

For example, to set the format to `uint16`, type the following. (Property names, like `Format`, are not case sensitive.)

```
m.format = 'uint16'
m =
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: true
      Offset: 1024
      Format: 'uint16'
      Repeat: Inf
        Data: 19488x1 uint16 array
```

Further read and write operations to the region mapped by `m` now treat the data in the file as a sequence of unsigned 16-bit integers. Whenever you change the value of a `memmapfile` property, MATLAB remaps the file to memory.

Selecting the File to Map

`filename` is the only required argument when you call the `memmapfile` constructor. When you call the `memmapfile` constructor, MATLAB assigns the file name that you specify to the `Filename` property of the new object instance.

Specify the file name as a quoted string, (e.g., `'records.dat'`). It must be first in the argument list and not specified as a parameter-value pair. `filename` must include a file name extension if the name of the file being mapped has an extension. The `filename` argument cannot include any wildcard characters (e.g., `*` or `?`), and is not case sensitive.

Note Unlike the other `memmapfile` constructor arguments, you must specify `filename` as a single string, and not as a parameter-value pair.

If the file to be mapped resides somewhere on the MATLAB path, you can use a partial pathname. If the path to the file is not fully specified, MATLAB searches for the file in your current working directory first, and then on the MATLAB path.

Once `memmapfile` locates the file, MATLAB stores the absolute path name for the file internally, and then uses this stored path to locate the file from that

point on. This enables you to work in other directories outside your current work directory and retain access to the mapped file.

You can change the value of the `Filename` property at any time after constructing the `memmapfile` object. You might want to do this if:

- You want to use the same `memmapfile` object on more than one file.
- You save your `memmapfile` object to a MAT-file, and then later load it back into MATLAB in an environment where the mapped file has been moved to a different location. This requires that you modify the path segment of the `Filename` string to represent the new location.

For example, save `memmapfile` object `m` to file `mymap.mat`:

```
disp(m.Filename)
    d:\matlab\mfiles\records.dat

save mymat m
```

Now move the file to another location, load the object back into MATLAB, and update the path in the `Filename` property:

```
load mymat m
m.Filename = 'f:\testfiles\oct1\records.dat'
```

Note You can only map an existing file. You cannot create a new file and map that file to memory in one operation. Use the MATLAB file I/O functions to create the file before attempting to map it to memory.

Setting the Start of the Mapped Region

By default, MATLAB begins a memory map at the start of the file. To begin the mapped region at some point beyond the start of the file, specify an `Offset` parameter in the call to the `memmapfile` constructor:

```
objname = memmapfile(filename, 'Offset', bytecount)
```

The `bytecount` value is the number of bytes from the beginning of the file to the point in the file where you want the memory map to start (a zero-based

offset). To map the file `records.dat` from a point 1024 bytes from the start and extending to the end of the file, type

```
m = memmapfile('records.dat', 'Offset', 1024);
```

You can change the starting position of an existing memory map by setting the `Offset` property for the associated object to a new value. The following command sets the offset of `memmapfile` object `m` to be 2,048 bytes from the start of the mapped file:

```
m.Offset = 2048;
```

Note The Sol64 platform supports aligned data access only. If you attempt to use a `memmapfile` offset on Sol64 that does not take the necessary alignment considerations into account, MATLAB generates an error. (See "Aligned Access on Sol64" on page 6-83).

Identifying the Contents of the Mapped Region

By default, MATLAB considers all the data in a mapped file to be a sequence of unsigned 8-bit integers. To have the data interpreted otherwise as it is read or written to in the mapped file, specify a `Format` parameter and value in your call to the constructor:

```
objname = memmapfile(filename, 'Format', formatspec)
```

The `formatspec` argument can either be a character string that identifies a single class used throughout the mapped region, or a cell array that specifies more than one class.

For example, say that you map a file that is 12 kilobytes in length. Data read from this file could be treated as a sequence of 6,000 16-bit (2-byte) integers, or as 1,500 8-byte double-precision floating-point numbers, to name just a couple of possibilities. Or you could read this data in as a combination of different types: for example, as 4,000 8-bit (1-byte) integers followed by 1,000 64-bit (8-byte) integers. You determine how MATLAB will interpret the mapped data by setting the `Format` property of the `memmapfile` object when you call its constructor.

MATLAB arrays are stored on disk in column-major order. (The sequence of array elements is column 1, row 1; column 1, row 2; column 1, last row; column 2, row 1, and so on.) You might need to transpose or rearrange the order of array elements when reading or writing via a memory map.

Note The Sol64 platform supports aligned data access only. If you attempt to use a `memmapfile` format on Sol64 that does not take the necessary alignment considerations into account, MATLAB generates an error. (See "Aligned Access on Sol64" on page 6-83).

For a list of data types supported for the `Format` property, see "Supported Data Types for the Format Property" on page 6-97.

For more information on format options see these sections:

- "Mapping a Single Data Type" on page 6-92
- "Formatting the Mapped Data to an Array" on page 6-93
- "Mapping Multiple Data Types and Arrays" on page 6-94

Mapping a Single Data Type. If the file region being mapped contains data of only one type, specify the `Format` value as a character string identifying that type:

```
objname = memmapfile(filename, 'Format', datatype)
```

The following command constructs a `memmapfile` object for the entire file `records.dat`, and sets the `Format` property for that object to `uint64`. Any read or write operations made via the memory map will read and write the file contents as a sequence of unsigned 64-bit integers:

```
m = memmapfile('records.dat', 'Format', 'uint64')
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: false
      Offset: 0
      Format: 'uint64'
      Repeat: Inf
        Data: 5000x1 uint64 array
```

You can change the value of the Format property at any time after the memmapfile object is constructed. Use the object.property syntax shown here in assigning the new value:

```
m.Format = 'int32';
```

Further read and write operations to the region mapped by m now treat the data in the file as a sequence of signed 32-bit integers.

Property names, like Format, are not case sensitive.

Formatting the Mapped Data to an Array. You can also specify an array shape to be applied to the data read or written to the mapped file, and a field name to be used in referencing this array. Use a cell array to hold these values either when calling the memmapfile constructor or when modifying m.Format after the object has been constructed. The cell array contains three elements: the class to be applied to the mapped region, the dimensions of the array shape that is applied to the region, and a field name to use in referencing the data:

```
objname = memmapfile(filename, ...
            'Format', {datatype, dimensions, varname})
```

The following command constructs a memmapfile object for a region of records.dat such that MATLAB handles the contents of the region as a 4-by-10-by-18 array of unsigned 32-bit integers, which you can reference in the structure of the returned object using the field name x:

```
m = memmapfile('records.dat',   ...
      'Offset', 1024,           ...
      'Format', {'uint32' [4 10 18] 'x'})
m =
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: false
      Offset: 1024
      Format: {'uint32' [4 10 18] 'x'}
      Repeat: Inf
        Data: 13x1 struct array with fields:
                x

A = m.Data(1).x;
```

```
whos A
  Name      Size                                    Bytes  Class

  A        4x10x18                                  2880   uint32 array

Grand total is 720 elements using 2880 bytes
```

You can change the class, array shape, or field name that MATLAB applies to the mapped region at any time by setting a new value for the `Format` property of the object:

```
m.Format = {'uint64' [30 4 10] 'x'};
A = m.Data(1).x;

whos A
  Name      Size                                    Bytes  Class

  A        30x4x10                                   9600  uint64 array

Grand total is 1200 elements using 9600 bytes
```

Mapping Multiple Data Types and Arrays. If the region being mapped is composed of segments of varying classes or array shapes, you can specify an individual format for each segment using an N-by-3 cell array, where N is the number of segments. The cells of each cell array row identify the class for that segment, the array dimensions to map the data to, and a field name by which to reference that segment:

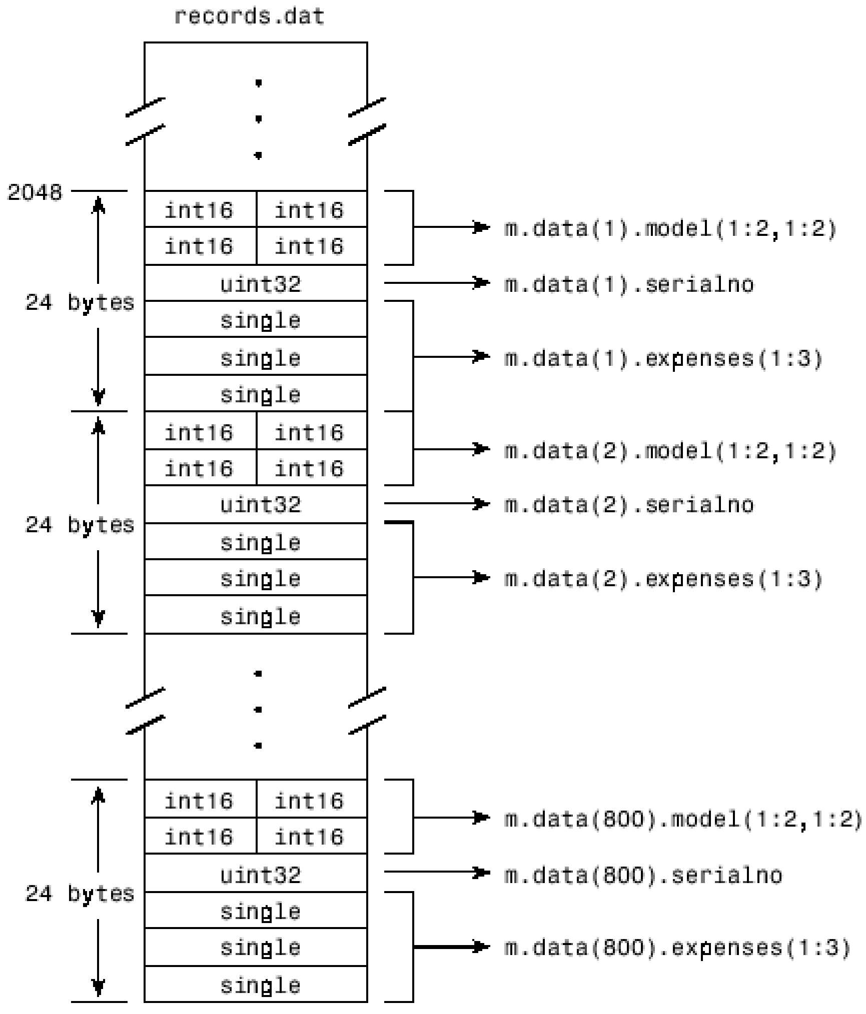
```
objname = memmapfile(filename,                              ...
            'Format', {                                     ...
               datatype1, dimensions1, fieldname1;          ...
               datatype2, dimensions2, fieldname2;          ...
                   :          :            :                ...
               datatypeN, dimensionsN, fieldnameN})
```

The following command maps a 24-kilobyte file containing data of three different classes: `int16`, `uint32`, and `single`. The `int16` data is mapped as a 2-by-2 matrix that can be accessed using the field name `model`. The `uint32` data is a scalar value accessed as field `serialno`. The `single` data is a 1-by-3 matrix named `expenses`.

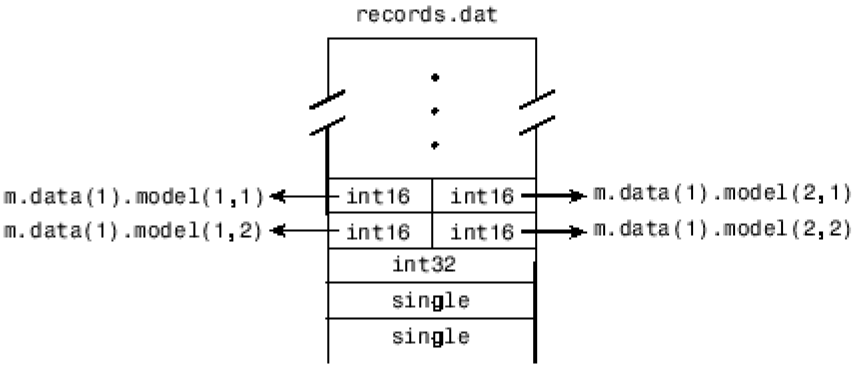
Each of these fields belongs to the 800-by-1 structure array `m.Data`:

```
m = memmapfile('records.dat',         ...
       'Offset', 2048,                ...
       'Format', {                    ...
          'int16'  [2 2] 'model';     ...
          'uint32' [1 1] 'serialno';  ...
          'single' [1 3] 'expenses'});
```

Mapping of the Example File



The figure below shows the ordering of the array elements more closely. In particular, it illustrates that MATLAB arrays are stored on the disk in column-major order. The sequence of array elements in the mapped file is row 1, column 1; row 2, column 1; row 1, column 2; and row 2, column 2.



If the data in your file is not stored in this order, you might need to transpose or rearrange the order of array elements when reading or writing via a memory map.

Supported Data Types for the Format Property. You can use any of the following classes when you specify a `Format` value. The default type is `uint8`.

Format String	Data Type Description
`'int8'`	Signed 8-bit integers
`'int16'`	Signed 16-bit integers
`'int32'`	Signed 32-bit integers
`'int64'`	Signed 64-bit integers
`'uint8'`	Unsigned 8-bit integers
`'uint16'`	Unsigned 16-bit integers
`'uint32'`	Unsigned 32-bit integers
`'uint64'`	Unsigned 64-bit integers
`'single'`	32-bit floating-point
`'double'`	64-bit floating-point

Repeating a Format Scheme

After you set a Format value for the memmapfile object, you can have MATLAB apply that format to the file data multiple times by specifying a Repeat value when you call the memmapfile constructor:

```
objname = memmapfile(filename,    ...
             'Format', formatspec, ...
             'Repeat', count)
```

The Repeat value applies to the whole format specifier, whether that specifier describes just a single class that repeats, or a more complex format that includes various classes and array shapes. The default Repeat value is infinity (inf), which means that the full extent of the Format specifier repeats as many times as possible within the mapped region.

The next example maps a file region identical to that of the previous example, except the pattern of int16, uint32, and single classes is repeated only three times within the mapped region of the file:

```
m = memmapfile('records.dat',         ...
       'Offset', 2048,                ...
       'Format', {                    ...
          'int16'  [2 2] 'model';      ...
          'uint32' [1 1] 'serialno';   ...
          'single' [1 3] 'expenses'},  ...
       'Repeat', 3);
```

You can change the value of the Repeat property at any time. To change the repeat value to 5, type

```
m.Repeat = 5;
```

Property names, like Repeat, are not case sensitive.

Keeping the Repeated Format Within the Mapped Region. MATLAB maps only the *full* pattern specified by the Format property. If you repeat a format such that it would cause the map to extend beyond the end of the file, then either of two things can happen:

- If you specify a repeat value of Inf, MATLAB applies to the map only those repeated segments that fit within the file in their entirety.

- If you specify a repeat value other than `Inf`, and that value would cause the map to extend beyond the end of the file, MATLAB generates an error.

Considering the last example, if the part of the file from `m.Offset` to the end were 70 bytes (instead of the 72 bytes required to repeat `m.Format` three times) and you used a `Repeat` value of `Inf`, then only two full repetitions of the specified format would have been mapped. The end result is as if you had constructed the map with this command:

```
m = memmapfile('records.dat',         ...
      'Offset', 2048,                 ...
      'Format', {                     ...
         'int16'  [2 2] 'model';      ...
         'uint32' [1 1] 'serialno';   ...
         'single' [1 3] 'expenses'},  ...
      'Repeat', 2);
```

If `Repeat` were set to 3 and you had only 70 bytes to the end of the file, you would get an error.

Note `memmapfile` does not expand or append to a mapped file. Use standard file I/O functions like `fopen` and `fwrite` to do this.

Setting the Type of Access

You can map a file region to allow either read-only or read and write access to its contents. Pass a `Writable` parameter and value in the `memmapfile` constructor, or set `m.Writable` on an existing object to set the type of access allowed:

```
objname = memmapfile(filename, 'Writable', trueorfalse)
```

The value passed can be either `true` (equal to `logical(1)`) or `false` (equal to `logical(0)`). By default, it is `false`, meaning that the mapped region is read only.

To map a read and write region of the file `records.dat` in memory, type

```
m = memmapfile('records.dat', 'Writable', true);
```

Note To successfully modify the file you are mapping to, you must have write permission for that file. If you do not have write permission, you can still set the `Writable` property to `true`, but attempting to write to the file generates an error.

You can change the value of the `Writable` property at any time. To make the memory map to `records.dat` read only, type:

```
m.Writable = false;
```

Property names, like `Writable`, are not case sensitive.

Reading a Mapped File

The most commonly used property of the `memmapfile` class is the `Data` property. It is through this property of the memory-map object that MATLAB provides all read and write access to the contents of the mapped file.

The actual mapping of a file to the MATLAB address space does not take place when you construct a `memmapfile` object. A memory map, based on the information currently stored in the mapped object, is generated the first time you reference or modify the `Data` property for that object.

After you map a file to memory, you can read the contents of that file using the same MATLAB statements used to read variables from the MATLAB workspace. By accessing the `Data` property of the memory map object, the contents of the mapped file appear as if they were an array in the currently active workspace. You simply index into this array to read the desired data from the file.

This section covers the following topics:

- "Example 4 — Modifying Map Parameters" on page 6-104

Improving Performance

MATLAB accesses data in structures more efficiently than it does data contained in objects. The main reason is that structures do not require the extra overhead of a `subsref` routine. Instead of reading directly from the `memmapfile` object, as shown here:

```
for k = 1 : N
   y(k) = m.Data(k);
end
```

you will get better performance when you assign the `Data` field to a variable, and then read or write the mapped file through this variable, as shown in this second example:

```
dataRef = m.Data;
for k = 1 : N
   y(k) = dataRef(k);
end
```

Example 1 — Reading a Single Data Type

This example maps a file of 100 double-precision floating-point numbers to memory. The map begins 1024 bytes from the start of the file, and ends 800 bytes (8 bytes per `double` times a `Repeat` value of 100) from that point.

If you haven't done so already, generate a test data file for use in the following examples by executing the `gendatafile` function defined under "Constructing a memmapfile Object" on page 6-86:

```
gendatafile('records.dat', 5000);
```

Now, construct the `memmapfile` object `m`, and show the format of its `Data` property:

```
m = memmapfile('records.dat', 'Format', 'double', ...
      'Offset', 1024, 'Repeat', 100);
```

```
d = m.Data;

whos d
  Name      Size                         Bytes  Class

  d       100x1                            800  double array

Grand total is 100 elements using 800 bytes
```

Read a selected set of numbers from the file by indexing into the single-precision array `m.Data`:

```
d(15:20)
ans =
  1.0e+009 *
    3.6045
    2.7006
    0.5745
    0.8896
    2.6079
    2.7053
```

Example 2 — Formatting File Data as a Matrix

This example is similar to the last, except that the constructor of the `memmapfile` object now specifies an array shape of 4-by-6 to be applied to the data as it is read from the mapped file. MATLAB maps the file contents into a structure array rather than a numeric array, as in the previous example:

```
m = memmapfile('records.dat',             ...
      'Format', {'double', [4 6], 'x'},  ...
      'Offset', 1024, 'Repeat', 100);

d = m.Data;

whos d
  Name      Size                         Bytes  Class

  d       100x1                          25264  struct array

Grand total is 2500 elements using 25264 bytes
```

When you read an element of the structure array, MATLAB presents the data in the form of a 4-by-6 array:

```
d(5).x
ans =
  1.0e+009 *
    3.1564    0.6684    2.1056    1.9357    1.2773    4.2219
    2.9520    0.8208    3.5044    1.7705    0.2112    2.3737
    1.4865    1.8144    1.9790    3.8724    2.9772    1.7183
    0.7131    3.6764    1.9643    0.0240    2.7922    0.8538
```

To index into the structure array field, use:

```
d(5).x(3,2:6)
ans =
  1.0e+009 *
    1.8144    1.9790    3.8724    2.9772    1.7183
```

Example 3 — Reading Multiple Data Types

This example maps a file containing more than one class. The different classes contained in the file are mapped as fields of the returned structure array `m.Data`.

The `Format` parameter passed in the constructor specifies that the first 80 bytes of the file are to be treated as a 5-by-8 matrix of `uint16`, and the 160 bytes after that as a 4-by-5 matrix of `double`. This pattern repeats until the end of the file is reached. The example shows different ways of reading the `Data` property of the object.

Start by calling the `memmapfile` constructor to create a memory map object, `m`:

```
m = memmapfile('records.dat',  ...
      'Format', {              ...
         'uint16' [5 8] 'x';   ...
         'double' [4 5] 'y' });
```

If you examine the `Data` property, MATLAB shows a 166-element structure array with two fields, one for each format specifier in the constructor:

```
d = m.Data
```

```
ans =
166x1 struct array with fields:
    x
    y
```

Examine one structure in the array to show the format of each field:

```
d(3)
ans =
    x: [5x8 uint16]
    y: [4x5 double]
```

Now read the `x` and `y` fields of that structure from the file. MATLAB formats the first block of data as a 5-by-8 matrix of `uint16`, as specified in the `Format` property, and the second block as a 4-by-5 matrix of `double`:

```
d(3).x
ans =
  34432  47500  19145  16868  38165  47956  35550  16853
  60654  51944  16874  47166  35397  58072  16850  56576
  51075  16876  12471  34369   8341  16853  44509  57652
  16863  16453   6666  11480  16869  58695  36217   5932
  57883  15551  41755  16874  37774  31693  54813  16865

d(3).y
ans =
  1.0e+009 *
    3.1229    1.5909    2.9831    2.2445    1.1659
    1.3284    3.0182    2.6685    3.7802    1.0837
    3.6013    2.3475    3.4137    0.7428    3.7613
    2.4399    1.9107    4.1096    4.2080    3.1667
```

Example 4 — Modifying Map Parameters

This example plots the Fourier transform output of data read from a file via a memory map. It then modifies several parameters of the existing map, reads from a different part of the data file, and plots a histogram from that data.

Create a memory-mapped object, mapping 1,000 elements of type `double` starting at the 1025th byte:

```
m = memmapfile('mybinary.bin', 'Offset', 1024,  ...
      'Format', 'double', 'Repeat', 1000);
```

Get data associated with the map and plot the FFT of the first 1000 values of the map. This is when the map is actually created, because no data has been referenced until this point:

```
plot(abs(fft(m.Data(1:1000))));
```

Get information about the memory map:

```
mapStruct = get(m)

mapStruct =
    Filename: 'd:\matlab\mfiles\mybinary.bin'
    Writable: 0
      Offset: 1024
      Format: 'double'
      Repeat: 1000
        Data: [1000x1 double]
```

Change the map, but continue using the same file:

```
m.Offset = 4096;
m.Format = 'single';
m.Repeat = 800;
```

Read from a different area of the file, and plot a histogram of the data. This maps a new region and unmaps the previous region:

```
hist(m.Data)
```

Writing to a Mapped File

Writing to a mapped file is done with standard MATLAB subscripted assignment commands. To write to a particular location in the file mapped to `memmapfile` object `m`, assign the value to the `m.Data` structure array index and field that map to that location.

If you haven't done so already, generate a test data file for use in the following examples by executing the gendatafile function defined under "Constructing a memmapfile Object" on page 6-86:

```
gendatafile('records.dat', 5000);
```

Now call the memmapfile constructor to create the object:

```
m = memmapfile('records.dat',  ...
      'Format', {               ...
         'uint16' [5 8] 'x';    ...
         'double' [4 5] 'y' });
```

If you are going to modify the mapped file, be sure that you have write permission, and that you set the Writable property of the memmapfile object to true (logical 1):

```
m.Writable = true;
```

Note You do not have to set Writable as a separate command, as done here. You can include a Writable parameter-value argument in the call to the memmapfile constructor.

Read from the 5-by-8 matrix x at m.Data(2):

```
d = m.Data;

d(2).x
ans =
  35330   4902  31861  16877  23791  61500  52748  16841
  51314  58795  16860  43523   8957   5182  16864  60110
  18415  16871  59373  61001  52007  16875  26374  28570
  16783   4356  52847  53977  16858  38427  16067  33318
  65372  48883  53612  16861  18882  39824  61529  16869
```

Update all values in that matrix using a standard MATLAB assignment statement:

```
d(2).x = d(2).x * 1.5;
```

Verify the results:

```
d(2).x
ans =
  52995   7353  47792  25316  35687  65535  65535  25262
  65535  65535  25290  65285  13436   7773  25296  65535
  27623  25307  65535  65535  65535  25313  39561  42855
  25175   6534  65535  65535  25287  57641  24101  49977
  65535  65535  65535  25292  28323  59736  65535  25304
```

This section covers the following topics:

- "Dimensions of the Data Field" on page 6-107
- "Writing Matrices to a Mapped File" on page 6-108
- "Selecting Appropriate Data Types" on page 6-111
- "Working with Copies of the Mapped Data" on page 6-111
- "Invalid Syntax for Writing to Mapped Memory" on page 6-112

Dimensions of the Data Field

The dimensions of a `memmapfile` object's `Data` field are set at the time you construct the object and cannot be changed. This differs from other MATLAB arrays that have dimensions you can modify using subscripted assignment.

For example, you can add a new column to the field of a MATLAB structure:

```
A.s = ones(4,5);

A.s(:,6) = [1 2 3 4];         % Add new column to A.s
size(A.s)
ans =
     4     6
```

But you cannot add a new column to a similar field of a structure that represents data mapped from a file. The following assignment to `m.Data(60).y` does not expand the size of `y`, but instead generates an error:

```
m.Data(60)
ans =
```

```
       x: [5x8 uint16]
       y: [4x5 double]

m.Data(60).y(:,6) = [1 2 3 4];        % Generates an error.
```

Thus, if you map an entire file and then append to that file after constructing the map, the appended data is not included in the mapped region. If you need to modify the dimensions of data that you have mapped to a `memmapfile` object, you must either modify the `Format` or `Repeat` properties for the object, or reconstruct the object.

Examples. Two examples of statements that attempt to modify the dimensions of a mapped `Data` field are shown here. These statements result in an error.

The first example attempts to diminish the size of the array by removing a row from the mapped array `m.Data`.

```
m.Data(5) = [];
```

The second example attempts to expand the size of a 50-row mapped array `x` by adding another row to it:

```
m.Data(2).x(1:51,31) = 1:51;
```

Writing Matrices to a Mapped File

The syntax to use when writing to mapped memory can depend on what format was used when you mapped memory to the file.

When Memory Is Mapped in Nonstructure Format. When you map a file as a sequence of a single class (e.g., a sequence of `uint16`), you can use the following syntax to write matrix `X` to the file:

```
m.Data = X;
```

This statement is valid only if all of the following conditions are true:

- The file is mapped as a sequence of elements of the same class, making `m.Data` an array of a nonstructure type.
- The class of `X` is the same as the class of `m.Data`.

- The number of elements in X equals the number of elements in m.Data.

This example maps a file as a sequence of 16-bit unsigned integers, and then uses the syntax shown above to write a matrix to the file. Map only a small part of the file, using a uint16 format for this segment:

```
m = memmapfile('records.dat', 'Writable', true', ...
    'Offset', 2000, 'Format', 'uint16', 'Repeat', 15);
```

Create a matrix X of the same size and write it to the mapped part of the file:

```
X = uint16(5:5:75);    % Sequence of 5 to 75, counting by fives.
m.data = X;
```

Verify that new values were written to the file:

```
m.offset = 1980;   m.repeat = 35;
reshape(m.data,5,7)'
ans =
   29158  16841  32915  37696    421      % <== At offset 1980
   16868  51434  17455  30645  16871
       5     10     15     20     25      % <== At offset 2000
      30     35     40     45     50
      55     60     65     70     75
   16872  50155  51100  26469  16873
   56776   6257  28746  16877  34374
```

When Memory Is Mapped in Scalar Structure Format. When you map a file as a sequence of a single class (e.g., a sequence of uint16), you can use the following syntax to write matrix X to the file:

```
m.Data.f = X;
```

This statement is valid only if all of the following conditions are true:

- The file is mapped as containing multiple classes that do not repeat, making m.Data a scalar structure.
- The class of X is the same as the class of m.Data.f.
- The number of elements in X equals that of m.Data.f.

This example maps a file as a 300-by-8 matrix of type `uint16` followed by a 200-by-5 matrix of type `double`, and then uses the syntax shown above to write a matrix to the file.

```
m = memmapfile('records.dat',     ...
      'Format', {                 ...
         'uint16' [300 8] 'x';    ...
         'double' [200 5] 'y' },  ...
      'Repeat', 1, 'Writable', true);

m.Data.x = ones(300, 8, 'uint16');
```

When Memory Is Mapped in Nonscalar Structure Format. When you map a file as a repeating sequence of multiple classes, you can use the following syntax to write matrix `X` to the file, providing that `k` is a scalar index:

```
m.Data(k).field = X;
```

To do this, the following conditions must be true:

- The file is mapped as containing multiple classes that can repeat, making `m.Data` a nonscalar structure.
- `k` is a scalar index.
- The class of `X` is the same as the class of `m.Data(k).field`.
- The number of elements in `X` equals that of `m.Data(k).field`.

This example maps a file as a matrix of type `uint16` followed by a matrix of type `double` that repeat 20 times, and then uses the syntax shown above to write a matrix to the file.

```
m = memmapfile('records.dat',    ...
      'Format', {                ...
         'uint16' [25 8] 'x';    ...
         'double' [15 5] 'y' },  ...
      'Repeat', 20, 'Writable', true);

d = m.Data;

d(12).x = ones(25,8,'uint16');
```

You can write to specific elements of field `x` as shown here:

```
d(12).x(3:5,1:end) = uint16(500);
d(12).x(3:5,1:end)
ans =
    500     500     500     500     500     500     500     500
    500     500     500     500     500     500     500     500
    500     500     500     500     500     500     500     500
```

Selecting Appropriate Data Types

All of the usual MATLAB indexing and class rules apply when assigning values to data via a memory map. The class that you assign to must be big enough to hold the value being assigned. For example,

```
m = memmapfile('records.dat', 'Format', 'uint8', ...
      'Writable', true);

d = m.Data;
d(5) = 300;
```

saturates the `x` variable because `x` is defined as an 8-bit integer:

```
d(5)
ans =
   255
```

Working with Copies of the Mapped Data

In the following code, the data in variable `block2` is a *copy* of the file data mapped by `m.Data(2)`. Because it is a copy, modifying array data in `block2` does not modify the data contained in the file:

First, destroy the `memmapfile` object and restore the test file `records.dat`, since you modified it by running the previous examples:

```
clear m
gendatafile('records.dat',50000);
```

Map the file as a series of `uint16` and `double` matrices and make a copy of `m.Data(2)` in `block2`:

```
m = memmapfile('records.dat',   ...
       'Format', {              ...
          'uint16' [5 8] 'x';   ...
          'double' [4 5] 'y' });

d = m.Data;
```

Write all zeros to the copy:

```
d(2).x(1:5,1:8) = 0;

d(2).x
ans =
      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
```

Verify that the data in the mapped file is not changed even though the copy of `m.Data(2).x` is written with zeros:

```
m.Data(2).x
ans =
  35330   4902  31861  16877  23791  61500  52748  16841
  51314  58795  16860  43523   8957   5182  16864  60110
  18415  16871  59373  61001  52007  16875  26374  28570
  16783   4356  52847  53977  16858  38427  16067  33318
  65372  48883  53612  16861  18882  39824  61529  16869
```

Invalid Syntax for Writing to Mapped Memory

Although you can expand the dimensions of a typical MATLAB array by assigning outside its current dimensions, this does not apply to the `Data` property of a `memmapfile` object. The following operation is invalid if `m.Data` has only 100 elements:

```
m.Data(120) = x;
```

If you need to expand the size of the mapped data region, first extend the map by updating the `Format` or `Repeat` property of the `memmapfile` object to reflect the new structure of the data in the mapped file.

Methods of the memmapfile Class

You can use the following methods on objects constructed from the `memmapfile` class.

Syntax	Description
`disp`	Displays properties of the object. The display does not include the object's name.
`get(obj)`	Returns the values of all properties of the `memmapfile` object in a structure array.
`get(obj, property)`	Returns the value of the specified property. `property` can be a string or cell array of strings, each containing a property name.

Using the disp Method

Use the `disp` method to display all properties of a `memmapfile` object. The text displayed includes only the property value, and not the object name or the MATLAB response string, `ans =`.

Construct object `m`:

```
m = memmapfile('records.dat',         ...
       'Offset', 2048,                ...
       'Format', {                    ...
          'int16'  [2 2] 'model';     ...
          'uint32' [1 1] 'serialno';  ...
          'single' [1 3] 'expenses'});
```

and display all of its properties:

```
disp(m)
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: false
      Offset: 2048
```

```
        Format: {'int16' [2 2] 'model'
                 'uint32' [1 1] 'serialno'
                 'single' [1 3] 'expenses'}
        Repeat: Inf
          Data: 16581x1 struct array with fields:
              model
           serialno
           expenses
```

Using the get Method

You can use the `get` method of the `memmapfile` class to return information on any or all of the object's properties. Specify one or more property names to get the values of specific properties.

This example returns the values of the `Offset`, `Repeat`, and `Format` properties for a `memmapfile` object. Use the `get` method to return the specified property values in a 1-by-3 cell array, `m_props`:

```
m_props = get(m, {'Offset', 'Repeat', 'Format'})
m_props =
    [2048]    [Inf]    {3x3 cell}

m_props{3}
ans =
    'int16'     [1x2 double]    'model'
    'uint32'    [1x2 double]    'serialno'
    'single'    [1x2 double]    'expenses'
```

You also can choose to use the `objname.property` syntax:

```
m_props = {m.Offset, m.Repeat, m.Format}
m_props =
    [2048]    [Inf]    {3x3 cell}
```

To return the values for all properties with `get`, pass just the object name:

```
get(m)
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: 0
      Offset: 2048
```

```
    Format: {3x3 cell}
    Repeat: Inf
      Data: [16581x1 struct]
```

Deleting a Memory Map

It is not necessary to explicitly call a destructor method to clear a `memmapfile` object from memory when you no longer need it. MATLAB calls the destructor for you whenever you do any of the following:

- Reassign another value to the `memmapfile` object's variable
- Clear the object's variable from memory
- Exit the function scope in which the object was created

The Effect of Shared Data Copies On Performance

When you assign the `Data` field of the `memmapfile` object to a variable, MATLAB makes a shared data copy of the mapped data. This is very efficient as no memory actually gets copied. In the following statement, `memdat` is a shared data copy of the data mapped from the file:

```
memdat = m.Data;
```

When you finish using the mapped data, make sure to clear any variables that shared data with the mapped file before clearing the object itself. If you clear the object first, then the sharing of data between the file and dependent variables is broken, and the data assigned to such variables must be copied into memory before the object is destroyed. If access to the mapped file was over a network, then copying this data to local memory can take considerable time. So, if the statement shown above assigns data to the variable `memdat`, you should be sure to clear `memdat` before clearing `m` when you are finished with the object.

Note Keep in mind that the `memmapfile` object can be cleared in any of the three ways described under “Deleting a Memory Map” on page 6-115.

Memory-Mapping Demo

In this demonstration, two separate MATLAB processes communicate with each other by writing and reading from a shared file. They share the file by mapping part of their memory space to a common location in the file. A write operation to the memory map belonging to the first process can be read from the map belonging to the second, and vice versa.

One MATLAB process (running `send.m`) writes a message to the file via its memory map. It also writes the length of the message to byte 1 in the file, which serves as a means of notifying the other process that a message is available. The second process (running `answer.m`) monitors byte 1 and, upon seeing it set, displays the received message, puts it into uppercase, and echoes the message back to the sender.

The send Function

This function prompts you to enter a string and then, using memory-mapping, passes the string to another instance of MATLAB that is running the `answer` function.

Copy the `send` and `answer` functions to files `send.m` and `answer.m` in your current working directory. Begin the demonstration by calling `send` with no inputs. Next, start a second MATLAB session on the same machine, and call the `answer` function in this session. To exit, press **Enter**.

```
function send
% Interactively send a message to ANSWER using memmapfile class.

filename = fullfile(tempdir, 'talk_answer.dat');

% Create the communications file if it is not already there.
if ~exist(filename, 'file')
    [f, msg] = fopen(filename, 'wb');
    if f ~= -1
        fwrite(f, zeros(1,256), 'uint8');
        fclose(f);
    else
        error('MATLAB:demo:send:cannotOpenFile', ...
              'Cannot open file "%s": %s.', filename, msg);
    end
```

```
end

% Memory map the file.
m = memmapfile(filename, 'Writable', true, 'Format', 'uint8');

while true
    % Set first byte to zero, indicating a message is not
    % yet ready.
    m.Data(1) = 0;

    str = input('Enter send string (or RETURN to end): ', 's');

    len = length(str);
    if (len == 0)
        disp('Terminating SEND function.')
        break;
    end

    str = str(1:min(len, 255));  % Message limited to 255 chars.

    % Update the file via the memory map.
    m.Data(2:len+1) = str;
    m.Data(1)=len;

    % Wait until the first byte is set back to zero,
    % indicating that a response is available.
    while (m.Data(1) ~= 0)
        pause(.25);
    end

    % Display the response.
    disp('response from ANSWER is:')
    disp(char(m.Data(2:len+1))')
end
```

The answer Function

The `answer` function starts a server that, using memory-mapping, watches for a message from `send`. When the message is received, `answer` replaces the

message with an uppercase version of it, and sends this new message back to send.

To use answer, call it with no inputs:

```
function answer
% Respond to SEND using memmapfile class.

disp('ANSWER server is awaiting message');

filename = fullfile(tempdir, 'talk_answer.dat');

% Create the communications file if it is not already there.
if ~exist(filename, 'file')
    [f, msg] = fopen(filename, 'wb');
    if f ~= -1
        fwrite(f, zeros(1,256), 'uint8');
        fclose(f);
    else
        error('MATLAB:demo:answer:cannotOpenFile', ...
              'Cannot open file "%s": %s.', filename, msg);
    end
end

% Memory map the file.
m = memmapfile(filename, 'Writable', true, 'Format', 'uint8');

while true
    % Wait till first byte is not zero.
    while m.Data(1) == 0
        pause(.25);
    end

    % The first byte now contains the length of the message.
    % Get it from m.
    msg = char(m.Data(2:1+m.Data(1)))';

    % Display the message.
    disp('Received message from SEND:')
    disp(msg)
```

```
    % Transform the message to all uppercase.
    m.Data(2:1+m.Data(1)) = upper(msg);

    % Signal to SEND that the response is ready.
    m.Data(1) = 0;
end
```

Running the Demo

To see what the demonstration looks like when it is run, first, start two separate MATLAB sessions on the same computer system. Call the `send` function in one and the `answer` function in the other to create a map in each of the processes' memory to the common file:

```
% Run SEND in the first MATLAB session.
send
Enter send string (or RETURN to end):


% Run ANSWER in the second MATLAB session.
answer
ANSWER server is awaiting message
```

Next, enter a message at the prompt displayed by the `send` function. MATLAB writes the message to the shared file. The second MATLAB session, running the `answer` function, loops on byte 1 of the shared file and, when the byte is written by `send`, `answer` reads the message from the file via its memory map. The `answer` function then puts the message into uppercase and writes it back to the file, and `send` (waiting for a reply) reads the message and displays it:

```
% SEND writes a message and reads the uppercase reply.
Hello. Is there anybody out there?
response from ANSWER is:
HELLO.  IS THERE ANYBODY OUT THERE?
Enter send string (or RETURN to end):


% ANSWER reads the message from SEND.
Received message from SEND:
Hello.  Is there anybody out there?
```

send writes a second message to the file. answer reads it, put it into uppercase, and then writes the message to the file:

```
% SEND writes a second message to the shared file.
I received your reply.
response from ANSWER is:
I RECEIVED YOUR REPLY.
Enter send string (or RETURN to end): <Enter>
Terminating SEND function.
```

```
% ANSWER reads the second message.
Received message from SEND:
I received your reply.
```

Exchanging Files over the Internet

In this section...

Overview

MATLAB software provides functions for exchanging files over the Internet. You can exchange files using common protocols, such as File Transfer Protocol (FTP), Simple Mail Transport Protocol (SMTP), and HyperText Transfer Protocol (HTTP). In addition, you can create zip archives to minimize the transmitted file size, and also save and work with Web pages.

Downloading Web Content and Files

MATLAB provides two functions for downloading Web pages and files using HTTP: `urlread` and `urlwrite`. With the `urlread` function, you can read and save the contents of a Web page to a string variable in the MATLAB workspace. With the `urlwrite` function, you can save a Web page's content to a file.

Because it creates a string variable in the workspace, the `urlread` function is useful for working with the contents of Web pages in MATLAB. The `urlwrite` function is useful for saving Web pages to a local directory.

Note When using `urlread`, remember that only the HTML in that specific Web page is retrieved. The hyperlink targets, images, and so on are not retrieved.

If you need to pass parameters to a Web page, the `urlread` and `urlwrite` functions let you use HTTP `post` and `get` methods. For more information, see the `urlread` and `urlwrite` reference pages.

Example — Using the urlread Function

The following procedure demonstrates how to retrieve the contents of the Web page containing the Recent File list at the MATLAB Central File Exchange, `http://www.mathworks.com/matlabcentral/fileexchange/`. It assigns the results to a string variable, `recentFile`, and it uses the `strfind` function to search the retrieved content for a specific word:

1 Retrieve the Web page content with the `urlread` function:

```
recentFile =
urlread('http://www.mathworks.com/matlabcentral/fileexchange/
loadFileList.do?objectType=fileexchange&orderBy=date&srt3=0');
```

2 After retrieving the content, run the `strfind` function on the `recentFile` variable:

```
hits = strfind(recentFile,'Simulink');
```

If the file contains the word `Simulink`, MATLAB stores the matches in the `hits` variable.

While you can manually pass arguments using the URL, the `urlread` function also lets you pass parameters to a Web page using standard HTTP methods, including `post` and `form`. Using the HTTP `get` method, which passes parameters in the URL, the following code queries Google for the word `Simulink`:

```
s =
urlread('http://www.google.com/search','get',{'q','Simulink'})
```

For more information, see the `urlread` reference page.

Example — Using the urlwrite Function

The following example builds on the procedure in the previous section. This example still uses `urlread` and checks for a specific word, but it also uses `urlwrite` to save the file if it contains any matches:

```
% The urlread function loads the contents of the Web page into
the % MATLAB workspace.

recentFile =
urlread('http://www.mathworks.com/matlabcentral/fileexchange/
loadFileList.do?objectType=fileexchange&orderBy=date&srt3=0');

% The strfind function searches for the word "Simulink".
hits = strfind(recentFile,'Simulink');

% The if statement checks for any hits.
if ~isempty(hits)

% If there are hits, the Web page will be saved locally
% using the urlwrite function.

urlwrite('http://www.mathworks.com/matlabcentral/fileexchange/
loadFileList.do?objectType=fileexchange&orderBy=date&srt3=0',
'contains_simulink.html');
end;
```

MATLAB saves the Web page as `contains_simulink.html`.

Creating and Decompressing Zip Archives

Using the `zip` and `unzip` functions, you can compress and decompress files and directories. The `zip` function compresses files or directories into a zip archive. The `unzip` function decompresses zip archives.

Example — Using the zip Function

Again building on the example from previous sections, the following code creates a zip archive of the retrieved Web page:

```
% The urlread function loads the contents of the Web page into
the % MATLAB workspace.
recentFile =
urlread('http://www.mathworks.com/matlabcentral/fileexchange/
loadFileList.do?objectType=fileexchange&orderBy=date&srt3=0');

% The strfind function searches for the word "Simulink".
```

```
hits = strfind(recentFile,'Simulink');

% The if statement checks for any hits.
if ~isempty(hits)

% If there are hits, the Web page will be saved locally
% using the urlwrite function.
urlwrite('http://www.mathworks.com/matlabcentral/fileexchange/
loadFileList.do?objectType=fileexchange&orderBy=date&srt3=0',
'contains_simulink.html');

% The zip function creates a zip archive of the retrieved Web
page.
zip('simulink_matches.zip','contains_simulink.html');
end;
```

Sending E-Mail

To send an e-mail from MATLAB, use the `sendmail` function. You can also attach files to an e-mail, which lets you mail files directly from MATLAB. To use `sendmail`, you must first set up your e-mail address and your SMTP server information with the `setpref` function.

The `setpref` function defines two mail-related preferences:

- E-mail address: This preference sets your e-mail address that will appear on the message. Here is an example of the syntax:

  ```
  setpref('Internet','E_mail','youraddress@yourserver.com');
  ```

- SMTP server: This preference sets your outgoing SMTP server address, which can be almost any e-mail server that supports the Post Office Protocol (POP) or the Internet Message Access Protocol (IMAP). Here is an example of the syntax:

  ```
  setpref('Internet', 'SMTP_Server', 'mail.server.network');
  ```

You should be able to find your outgoing SMTP server address in your e-mail account settings in your e-mail client application. You can also contact your system administrator for the information.

Note The `sendmail` function does not support e-mail servers that require authentication.

Once you have properly configured MATLAB, you can use the `sendmail` function. The `sendmail` function requires at least two arguments: the recipient's e-mail address and the e-mail subject:

```
sendmail('recipient@someserver.com', 'Hello From MATLAB!');
```

You can supply multiple e-mail addresses using a cell array of strings, such as:

```
sendmail({'recipient@someserver.com', ...
'recipient2@someserver.com'}, 'Hello From MATLAB!');
```

You can also specify a message body with the `sendmail` function, such as:

```
sendmail('recipient@someserver.com', 'Hello From MATLAB!', ...
'Thanks for using sendmail.');
```

In addition, you can also attach files to an e-mail using the `sendmail` function, such as:

```
sendmail('recipient@someserver.com', 'Hello from MATLAB!', ...
'Thanks for using sendmail.', 'C:\yourFileSystem\message.txt');
```

You cannot attach a file without including a message. However, the message can be empty. You can also attach multiple files to an e-mail with the `sendmail` function, such as:

```
sendmail('recipient@someserver.com', 'Hello from MATLAB!', ...
'Thanks for using sendmail.', ...
{'C:\yourFileSystem\message.txt',...
'C:\yourFileSystem\message2.txt'});
```

Example — Using the sendmail Function

The following example sends e-mail with the retrieved Web page archive attached if it contains any matches for the specified word:

```
% The urlread function loads the contents of the Web page into
the  % MATLAB workspace.
recentFile =
urlread('http://www.mathworks.com/matlabcentral/fileexchange/
loadFileList.do?objectType=fileexchange&orderBy=date&srt3=0');

% The strfind function searches for the word "Simulink".
hits = strfind(recentFile,'Simulink');

% The if statement checks for any hits.
if ~isempty(hits)

% If there are hits, the Web page will be saved locally
% using the urlwrite function.
urlwrite('http://www.mathworks.com/matlabcentral/fileexchange/
loadFileList.do?objectType=fileexchange&orderBy=date&srt3=0',
'contains_simulink.html');

% The zip function creates a zip archive of the retrieved web
page.
zip('simulink_matches.zip','contains_simulink.html');

% The setpref function supplies your e-mail address and SMTP
% server address to MATLAB.
setpref('Internet','SMTP_Server','mail.server.network');
setpref('Internet', 'E_mail', 'youraddress@yourserver.com');

% The sendmail function sends an e-mail with the zip archive of
the
% retrieved Web page attached.
sendmail('youraddress@yourserver.com', 'New Simulink Files
Found', 'New Simulink files uploaded to MATLAB Central. See
attached zip archive.', 'simulink_matches.zip');
end;
```

Performing FTP File Operations

From MATLAB, you can connect to an FTP server to perform remote file operations. The following procedure uses a public MathWorks FTP server

(`ftp.mathworks.com`). To perform any file operation on an FTP server, follow these steps:

1 Connect to the server using the `ftp` function.

   For example, you can create an FTP object for the public MathWorks FTP server with `tmw=ftp('ftp.mathworks.com')`.

2 Perform the file operations using appropriate MATLAB FTP functions as methods acting on the server object.

   For example, you can display the file directories on the FTP server with `dir(tmw)`.

3 When you finish working on the server, close the connection object using the close function.

   For example, you can disconnect from the FTP server with `close(tmw)`.

Example — Retrieving a File from an FTP Server

In this example, you retrieve the file `pub/pentium/Moler_1.txt`, which is on the MathWorks FTP server. You can run this example; the FTP server and content are valid.

1 Connect to the MathWorks FTP server using `ftp`. This creates the server object `tmw`:

```
tmw=ftp('ftp.mathworks.com');
```

2 List the contents of the server using the FTP `dir` function, which operates on the server object `tmw`:

```
dir(tmw)
```

3 Change to the `pub` directory by using the FTP `cd` function. As with all FTP functions, you need to specify the server object you created using `tmw` as part of the syntax. In this case, this is `tmw`:

```
cd(tmw,'pub');
```

The server object `tmw` represents the current directory on the FTP server, which now is `pub`.

4 Now when you run:

```
dir(tmw)
```

you see the contents of `pub`, rather than the top level contents as displayed previously when you ran `dir(tmw)`.

5 Use `mget` to retrieve any of the files from the current directory on the FTP server to the MATLAB current directory:

```
mget(tmw,'filename');
```

6 Close the FTP connection using `close`.

```
close(tmw);
```

Summary of FTP Functions

The following table lists the available FTP functions. For more information, refer to the applicable reference pages.

Function	Description
`ascii`	Set FTP transfer type to ASCII (convert new lines).
`binary`	Set FTP transfer type to binary (transfer verbatim, default).
`cd (ftp)`	Change current directory on FTP server.
`delete (ftp)`	Delete file on FTP server.
`dir (ftp)`	List contents of directory on FTP server.
`close (ftp)`	Close connection with FTP server.
`ftp`	Connect to FTP server, creating an FTP object.
`mget`	Download file from FTP site.
`mkdir (ftp)`	Create new directory on FTP server.
`mput`	Upload file or directory to FTP server.

Function	Description
`rename`	Rename file on FTP server.
`rmdir (ftp)`	Remove directory on FTP server.

7

Scientific Data File Formats

This section describes how to import and export data in several standard scientific data formats. Topics covered include

Common Data Format (CDF) Files

In this section...

Note For information about working with netCDF files, which is a completely separate, incompatible format, see "Network Common Data Form (netCDF) Files " on page 7-8.

Getting Information About CDF Files

To get information about the contents of a Common Data Format (CDF) file, use the `cdfinfo` function. CDF was created by the National Space Science Data Center (NSSDC) to provide a self-describing data storage and manipulation format that matches the structure of scientific data and applications (i.e., statistical and numerical methods, visualization, and management). The `cdfinfo` function returns a structure containing general information about the file and detailed information about the variables and attributes in the file. For more information about this format, see the CDF Web site.

The following example returns information about the sample CDF file included with MATLAB. To determine the variables contained in the file, view the `Variables` field. This field contains a cell array that lists all the variables in the file with information that describes the variable, such as name, size, and data type. For an example, see "Importing Data from a CDF File" on page 7-3.

Note Because `cdfinfo` creates temporary files, make sure that your current working directory is writable before attempting to use the function.

```
info = cdfinfo('example.cdf')
```

```
info =

              Filename: 'example.cdf'
           FileModDate: '09-Mar-2001 16:45:22'
              FileSize: 1240
                Format: 'CDF'
         FormatVersion: '2.7.0'
          FileSettings: [1x1 struct]
              Subfiles: {}
             Variables: {5x6 cell}
      GlobalAttributes: [1x1 struct]
    VariableAttributes: [1x1 struct]
```

Importing Data from a CDF File

To import data into the MATLAB workspace from a Common Data Format (CDF) file, use the `cdfread` function. CDF was created by the National Space Science Data Center (NSSDC) to provide a self-describing data storage and manipulation format that matches the structure of scientific data and applications (i.e., statistical and numerical methods, visualization, and management). Using this function, you can import all the data in the file, specific variables, specific records, or subsets of the data in a specific variable. The following examples illustrate some of these capabilities.

1 To get information about the contents of a CDF file, such as the names of variables in the CDF file, use the `cdfinfo` function. In this example, the `Variables` field indicates that the file contains five variables. The first variable, `Time`, is made up of 24 records containing CDF epoch data. The next two variables, `Longitude` and `Latitude`, have only one associated record containing `int8` data. For details about how to interpret the data returned in the `Variables` field, see `cdfinfo`.

```
info = cdfinfo('example.cdf');

vars = info.Variables

vars =

  Columns 1 through 5

    'Time'                [1x2 double]   [24]   'epoch'    'T/'
```

```
 'Longitude'          [1x2 double]   [ 1]   'int8'     'F/FT'
 'Latitude'           [1x2 double]   [ 1]   'int8'     'F/TF'
 'Data'               [1x3 double]   [ 1]   'double'   'T/TTT'
 'multidimensional    [1x4 double]   [ 1]   'uint8'    'T/TTTT'

Column 6

  'Full'
  'Full'
  'Full'
  'Full'
  'Full'
```

2 To read all of the data in the CDF file, use the `cdfread` function. The function returns the data in a 24-by-5 cell array. The five columns of data correspond to the five variables; the 24 rows correspond to the 24 records associated with the Time variable and padding elements for the rows associated with the other variables. The padding value used is specified in the CDF file.

```
data = cdfread('example.cdf');

whos data
  Name       Size              Bytes  Class    Attributes

  data      24x5               14784  cell
```

3 To read the data associated with a particular variable, use the `'Variable'` parameter, specifying a cell array of variable names as the value of this parameter. Variable names are case sensitive. For example, the following code reads the Longitude and Latitude variables from the file. The return value `data` is a 24-by-2 cell array, where each cell contains `int8` data.

```
var_time = cdfread('example.cdf','Variable',{'Longitude','Latitude'});

whos var_time
  Name           Size            Bytes  Class    Attributes

  var_time      24x1              4608  cell
```

Speeding Up Read Operations

The `cdfread` function offers two ways to speed up read operations when working with large data sets:

- Reducing the number of elements in the returned cell array
- Returning CDF epoch values as MATLAB serial date numbers rather than as MATLAB `cdfepoch` objects

To reduce the number of elements in the returned cell array, specify the `'CombineRecords'` parameter. By default, `cdfread` creates a cell array with a separate element for every variable and every record in each variable, padding the records dimension to create a rectangular cell array. For example, reading all the data from the example file produces an output cell array, 24-by-5, where the columns represent variables and the rows represent the records for each variable. When you set the `'CombineRecords'` parameter to `true`, `cdfread` creates a separate element for each variable but saves time by putting all the records associated with a variable in a single cell array element. Thus, reading the data from the example file with `'CombineRecords'` set to `true` produces a 1-by-5 cell array, as shown below.

```
data_combined = cdfread('example.cdf','CombineRecords',true);

whos
  Name                 Size            Bytes  Class    Attributes

  data                24x5             14784  cell
  data_combined        1x5              2364  cell
```

When combining records, note that the dimensions of the data in the cell change. For example, if a variable has 20 records, each of which is a scalar value, the data in the cell array for the combined element contains a 20-by-1 vector of values. If each record is a 3-by-4 array, the cell array element contains a 20-by-3-by-4 array. For combined data, `cdfread` adds a dimension to the data, the first dimension, that is the index into the records.

Another way to speed up read operations is to read CDF epoch values as MATLAB serial date numbers. By default, `cdfread` creates a MATLAB `cdfepoch` object for each CDF epoch value in the file. If you specify the `'ConvertEpochToDatenum'` parameter, setting it to `true`, `cdfread` returns

CDF epoch values as MATLAB serial date numbers. For more information about working with MATLAB `cdfepoch` objects, see “Representing CDF Time Values” on page 7-6.

```
data_datenums = cdfread('example.cdf','ConvertEpochToDatenum',true);

whos
  Name                Size            Bytes  Class    Attributes

  data               24x5             14784  cell
  data_combined       1x5              2364  cell
  var_time           24x1              4608  cell
```

Representing CDF Time Values

CDF represents time differently than MATLAB. CDF represents date and time as the number of milliseconds since 1-Jan-0000. This is called an *epoch* in CDF terminology. MATLAB represents date and time as a serial date number, which is the number of days since 0-Jan-0000. To represent CDF dates, MATLAB uses an object called a CDF epoch object. To access the time information in a CDF object, use the object’s `todatenum` method.

For example, this code extracts the date information from a CDF epoch object:

1. Extract the date information from the CDF epoch object returned in the cell array `data` (see “Importing Data from a CDF File” on page 7-3). Use the `todatenum` method of the CDF epoch object to get the date information, which is returned as a MATLAB serial date number.

   ```
   m_date = todatenum(data{1});
   ```

2. View the MATLAB serial date number as a string.

   ```
   datestr(m_date)
   ans =

   01-Jan-2001
   ```

Exporting Data to a CDF File

To export data from the MATLAB workspace to a Common Data Format (CDF) file, use the `cdfwrite` function. CDF was created by the National

Space Science Data Center (NSSDC) to provide a self-describing data storage and manipulation format that matches the structure of scientific data and applications (i.e., statistical and numerical methods, visualization, and management). Using this function, you can write variables and attributes to the file, specifying their names and associated values. See the `cdfwrite` reference page for more information.

This example shows how to write date information to a CDF file. Note how the example uses the CDF epoch object constructor, `cdfepoch`, to convert a MATLAB serial date number into a CDF epoch.

```
cdfwrite('myfile',{'Time_val',cdfepoch(now)});
```

You can convert a `cdfepoch` object back into a MATLAB serial date number with the `todatenum` function.

Network Common Data Form (netCDF) Files

In this section...

Note For information about working with Common Data Format (CDF) files, which is a completely separate, incompatible format, see "Common Data Format (CDF) Files" on page 7-2.

Overview

MATLAB provides access to the routines in the netCDF C library that you can use to read data from netCDF files and write data to netCDF files. MATLAB provides this access through a set of MATLAB functions that correspond to the functions in the netCDF C library. MATLAB groups the functions into a package, called `netcdf`. To call one of the functions in the package, you must specify the package name. For a complete list of all the functions, see `netcdf`.

Note The MATLAB netCDF functions support netCDF Version 3.6.2.

This section does not attempt to describe all features of the netCDF library or explain basic netCDF programming concepts. To use the MATLAB netCDF functions effectively, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.. The following sections provide details about how to use the MATLAB netCDF functions.

Mapping netCDF API Syntax to MATLAB Function Syntax

For the most part, the MATLAB netCDF functions correspond directly to routines in the netCDF C library. For example, the MATLAB function `netcdf.open` corresponds to the netCDF library routine `nc_open`. In some cases, one MATLAB function corresponds to a group of netCDF library functions. For example, instead of creating MATLAB versions of every netCDF library `nc_put_att_`*`type`* function, where *`type`* represents a data type, MATLAB uses one function, `netcdf.putAtt`, to handle all supported data types.

The syntax of the MATLAB functions is similar to the netCDF library routines, with some exceptions. For example, the netCDF C library routines use input parameters to return data, while their MATLAB counterparts use one or more return values. For example, the following is the function signature of the `nc_open` routine in the netCDF library. Note how the netCDF file identifier is returned in the `ncidp` argument.

```
int nc_open (const char *path, int omode, int *ncidp); /* C syntax */
```

The following shows the signature of the corresponding MATLAB function, `netcdf.open`. Like its netCDF C library counterpart, the MATLAB netCDF function accepts a character string that specifies the file name and a constant that specifies the access mode. Note, however, that the MATLAB `netcdf.open` function returns the file identifier, `ncid`, as a return value.

```
ncid = netcdf.open(filename, mode)
```

To see a list of all the functions in the MATLAB netCDF package, see the `netCDF` reference page.

Mapping MATLAB Classes to netCDF Data Types

MATLAB attempts to map netCDF data types to the corresponding MATLAB class that best matches. For example, netCDF functions map the MATLAB `double` class to the netCDF `NC_DOUBLE` data type. The following table shows this mapping. For more information about netCDF data types, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

MATLAB Class	netCDF Data Type	Notes
int8	NC_BYTE	netCDF interprets byte data as either signed or unsigned.
uint8	NC_BYTE	netCDF interprets byte data as either signed or unsigned.
char	NC_CHAR	
int16	NC_SHORT	
uint16	No equivalent	
int32	NC_INT	
uint32	No equivalent	
int64	No equivalent	
uint64	No equivalent	
single	NC_FLOAT	
double	NC_DOUBLE	

Example: Exploring the Contents of a netCDF File

This example shows how to use the MATLAB netCDF functions to explore the contents of a netCDF file. The section uses the example netCDF file included with MATLAB, `example.nc`, as an illustration. For an example of reading data from a netCDF file, see "Example: Reading Data from a netCDF File" on page 7-14

1 Open the netCDF file using the `netcdf.open` function. This function returns an identifier that you use thereafter to refer to the file.

```
ncid = netcdf.open('example.nc','NC_NOWRITE');
```

2 Explore the contents of the file using the `netcdf.inq` function. This function returns the number of dimensions, variables, and global attributes in the file, and returns the identifier of the unlimited dimension in the file. The unlimited dimension can grow.

```
[ndims,nvars,natts,unlimdimID]= netcdf.inq(ncid)
```

```
ndims =

     4


nvars =

     4


natts =

     1


unlimdimID =

     3
```

3. Get more information about the dimensions, variables, and global attributes in the file by using netCDF inquiry functions. For example, to get information about the global attribute, first get the name of the attribute, using the `netcdf.inqAttName` function. After you get the name, `'creation_date'` in this case, you can use the `netcdf.inqAtt` function to get information about the data type and length of the attribute.

   To get the name of an attribute, you must specify the ID of the variable the attribute is associated with and the attribute number. To access a global attribute, which isn't associated with a particular variable, use the constant `'NC_GLOBAL'` as the variable ID. The attribute number is a zero-based index that identifies the attribute. For example, the first attribute has the index value `0`, and so on.

```
global_att_name = netcdf.inqAttName(ncid,netcdf.getConstant('NC_GLOBAL'),0)

global_att_name =

creation_date

[xtype attlen] = netcdf.inqAtt(ncid,netcdf.getConstant('NC_GLOBAL'),global_att_name)
```

```
xtype =

     2


attlen =

    11
```

4 Get the value of the attribute, using the `netcdf.getAtt` function.

```
global_att_value = netcdf.getAtt(ncid,netcdf.getConstant('NC_GLOBAL'),global_att_name)

global_att_value =

09-Jun-2008
```

5 Get information about the dimensions defined in the file through a series of calls to `netcdf.inqDim`. This function returns the name and length of the dimension. The `netcdf.inqDim` function requires the dimension ID, which is a zero-based index that identifies the dimensions. For example, the first dimension has the index value `0`, and so on.

```
[dimname, dimlen] = netcdf.inqDim(ncid,0)

dimname =

x

dimlen =

    50
```

The following table describes the dimensions in the example file.

Dimension Name	Dimension Length
x	50
y	50

Dimension Name	Dimension Length
z	5
t	0 (unlimited)

6 Get information about the variables in the file through a series of calls to `netcdf.inqVar`. This function returns the name, data type, dimension ID, and the number of attributes associated with the variable. The `netcdf.inqVar` function requires the variable ID, which is a zero-based index that identifies the variables. For example, the first variable has the index value `0`, and so on.

```
[varname, vartype, dimids, natts] = netcdf.inqVar(ncid,0)

varname =

avagadros_number


vartype =

     6


dimids =

     []


natts =

     1
```

The following table describes the variables in the example file. The data type information is the numeric value of the netCDF data type constants, such as, `NC_INT` and `NC_BYTE`. See the official netCDF documentation for information about these constants.

Variable Name	Variable Type	Variable Dimension IDs	Number of Attributes
avagadros_number	6	[]	1
temperature	3	0	4
peaks	5	[0 1]	1
time_series	4	[2 3]	1

Example: Reading Data from a netCDF File

After you understand the contents of a netCDF file, by using the inquiry functions, you can retrieve the data from the variables and attributes in the file. To read the data associated with the variable `avagadros_number` in the example file, use the `netcdf.getVar` function. The following example uses the netCDF file identifier returned in the previous section, "Example: Exploring the Contents of a netCDF File" on page 7-10. The variable ID is a zero-based index that identifies the variables. For example, the first variable has the index value `0`, and so on. (To learn how to write data to a netCDF file, see "Example: Storing Data in a netCDF File" on page 7-14.)

```
A_number = netcdf.getVar(ncid,0)

A_number =

  6.0221e+023
```

The netCDF functions automatically choose the MATLAB class that best matches the netCDF data type, but you can also specify the class of the return data by using an optional argument to `netcdf.getVar`.

Example: Storing Data in a netCDF File

To store data in a netCDF file, you can use the MATLAB netCDF functions to create a file, define dimensions in the file, create a variable in the file, and write data to the variable. Note that you must define dimensions in the file before you can create variables. To run the following example, you must have write permission in your current directory.

1 Create a variable in the MATLAB workspace. This example creates a 50-element vector of numeric values named `my_data`. The vector is of class `double`.

```
my_data = linspace(0,49,50);
```

2 Create a netCDF file (or open an existing file). The example uses the `netcdf.create` function to create a new file, named `my_file.nc`, and opens it for write access.

```
ncid = netcdf.create('my_file.nc','NC_WRITE');
```

When you create a netCDF file, the file opens in define mode. You must be in define mode to define dimensions and variables.

3 Define a dimension in the file, using the `netcdf.defDim` function. You must define dimensions in the file before you can define variables and write data to the file. When you define a dimension, you give it a name and a length. To create an unlimited dimension, i.e., a dimension that can grow, specify the constant `NC_UNLIMITED` in place of the dimension length.

```
dimid = netcdf.defDim(ncid,'my_dim',50);
```

4 Define a variable on the dimension, using the `netcdf.defVar` function. When you define a variable, you give it a name, data type, and a dimension ID. You must use one of the netCDF constants to specify the data type, listed in "Mapping MATLAB Classes to netCDF Data Types" on page 7-9.

```
varid = netcdf.defVar(ncid,'my_var','NC_BYTE',dimid);
```

5 Take the netCDF file out of define mode. To write data to a file, you must be in data mode.

```
netcdf.endDef(ncid);
```

6 Write the data from the MATLAB workspace into the variable in the netCDF file, using the `netcdf.putVar` function. Note that the data in the workspace is of class `double` but the variable in the netCDF file is of type `NC_BYTE`. The MATLAB netCDF functions automatically do the conversion.

```
netcdf.putVar(ncid,varid,my_data);
```

7 Close the file, using the `netcdf.close` function.

```
netcdf.close(ncid);
```

8 Verify that the data was written to the file by opening the file and reading the data from the variable into a new variable in the MATLAB workspace. Because the variable is the first variable in the file (and the only one), you can specify 0 (zero) for the variable ID—identifiers are zero-based indexes.

```
ncid2 = netcdf.open('my_file.nc','NC_NOWRITE');

data_in_file = netcdf.getVar(ncid2,0)

data_in_file =

      0
      1
      2
      3
      4
      5
      6
      7
      8
      9
      .
      .
      .
```

Because you stored the data in the file as `NC_BYTE`, MATLAB reads the data from the variable into the workspace as class `int8`.

Flexible Image Transport System (FITS) Files

In this section...

Getting Information About FITS Files

To get information about the contents of a Flexible Image Transport System (FITS) file, use the `fitsinfo` function. The FITS file format is the standard data format used in astronomy, endorsed by both NASA and the International Astronomical Union (IAU). For more information about the FITS standard, go to the official FITS Web site, `http://fits.gsfc.nasa.gov/`.

A data file in FITS format can contain multiple components, each marked by an ASCII text header followed by binary data. The first component in a FITS file is known as the *primary*, which can be followed by any number of other components, called *extensions,* in FITS terminology. The `fitsinfo` function returns a structure containing the information about the file and detailed information about the data in the file. This example returns information about a sample FITS file included with MATLAB. The structure returned contains fields for the primary component, `PrimaryData`, and all the extensions in the file, such as the `BinaryTable`, `Image`, and `AsciiTable` extensions.

```
info = fitsinfo('tst0012.fits')

info =

       Filename: 'tst0012.fits'
    FileModDate: '12-Mar-2001 18:37:46'
       FileSize: 109440
       Contents: {1x5 cell}
    PrimaryData: [1x1 struct]
    BinaryTable: [1x1 struct]
        Unknown: [1x1 struct]
          Image: [1x1 struct]
     AsciiTable: [1x1 struct]
```

Importing Data from a FITS File

To import data into the MATLAB workspace from a Flexible Image Transport System (FITS) file, use the fitsread function. The FITS file format is designed to store scientific data sets consisting of multidimensional arrays (1-D spectra, 2-D images, or 3-D data cubes) and two-dimensional tables containing rows and columns of data. Using this function, you can import the data in the PrimaryData section of the file or you can import the data in any of the extensions in the file, such as the Image extension. This example illustrates how to use the fitsread function to read data from a FITS file:

1 Determine which extensions the FITS file contains, using the fitsinfo function.

```
info = fitsinfo('tst0012.fits')

info =

       Filename: 'tst0012.fits'
    FileModDate: '12-Mar-2001 18:37:46'
       FileSize: 109440
       Contents: {1x5 cell}
    PrimaryData: [1x1 struct]
    BinaryTable: [1x1 struct]
        Unknown: [1x1 struct]
          Image: [1x1 struct]
     AsciiTable: [1x1 struct]
```

The info structure shows that the file contains several extensions including the BinaryTable, AsciiTable, and Image extensions.

2 Read data from the file.

To read the PrimaryData in the file, specify the filename as the only argument:

```
pdata = fitsread('tst0012.fits');
```

To read any of the extensions in the file, you must specify the name of the extension as an optional parameter. This example reads the BinaryTable extension from the FITS file:

```
bindata = fitsread('tst0012.fits','bintable');
```

Note To read the `BinaryTable` extension using `fitsread`, you must specify the parameter `'bintable'`. Similarly, to read the `AsciiTable` extension, you must specify the parameter `'table'`. See the `fitsread` reference page for more information.

Hierarchical Data Format (HDF5) Files

Note For information about working with HDF4 data, which is a completely separate, incompatible format, see "Hierarchical Data Format (HDF4) Files" on page 7-45.

Using the MATLAB High-Level HDF5 Functions

Hierarchical Data Format, Version 5, (HDF5) is a general-purpose, machine-independent standard for storing scientific data in files, developed by the National Center for Supercomputing Applications (NCSA). HDF5 is used by a wide range of engineering and scientific fields that want a standard way to store data so that it can be shared. For more information about the HDF5 file format, read the HDF5 documentation available at the HDF Web site (`http://www.hdfgroup.org`).

The MATLAB high-level HDF5 functions provide an easy way to import data or metadata from an HDF5 file, or write data to an HDF5 file. The following sections provide more detail about using these functions.

Determining the Contents of an HDF5 File

HDF5 files can contain data and metadata, called *attributes*. HDF5 files organize the data and metadata in a hierarchical structure similar to the hierarchical structure of a UNIX file system.

In an HDF5 file, the directories in the hierarchy are called *groups*. A group can contain other groups, data sets, attributes, links, and data types. A data set is a collection of data, such as a multidimensional numeric array or string. An attribute is any data that is associated with another entity, such as a data set. A link is similar to a UNIX file system symbolic link. Links are a way to reference data without having to make a copy of the data.

Data types are a description of the data in the data set or attribute. Data types tell how to interpret the data in the data set. For example, a file might contain a data type called “Reading” that is comprised of three elements: a longitude value, a latitude value, and a temperature value.

To explore the hierarchical organization of an HDF5 file, use the `hdf5info` function. For example, to find out what the sample HDF5 file, `example.h5`, contains, use this syntax:

```
fileinfo = hdf5info('example.h5');
```

`hdf5info` returns a structure that contains various information about the HDF5 file, including the name of the file and the version of the HDF5 library that MATLAB is using:

```
fileinfo =

          Filename: 'example.h5'
        LibVersion: '1.6.5'
            Offset: 0
          FileSize: 8172
    GroupHierarchy: [1x1 struct]
```

In the information returned by `hdf5info`, look at the `GroupHierarchy` field. This field is a structure that describes the top-level group in the file, called the *root* group. Using the UNIX convention, HDF5 names this top-level group `/` (forward slash), as shown in the `Name` field of the `GroupHierarchy` structure.

```
toplevel = fileinfo.GroupHierarchy

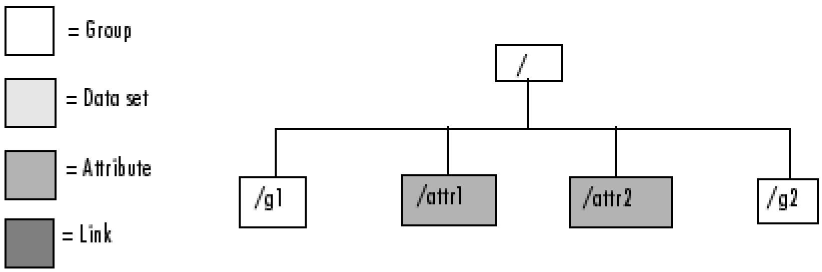
toplevel =

      Filename: 'C:\matlab\toolbox\matlab\demos\example.h5'
          Name: '/'
```

```
          Groups: [1x2 struct]
        Datasets: []
       Datatypes: []
           Links: []
      Attributes: [1x2 struct]
```

By looking at the Groups and Attributes fields, you can see that the file contains two groups and two attributes. The Datasets, Datatypes, and Links fields are all empty, indicating that the root group does not contain any data sets, data types, or links.

The following figure illustrates the organization of the root group in the sample HDF5 file example.h5.



Organization of the Root Group of the Sample HDF5 File

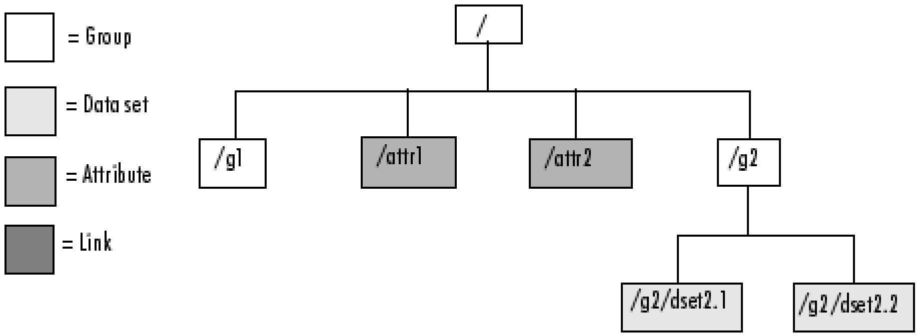
To explore the contents of the sample HDF5 file further, examine one of the two structures in the Groups field of the GroupHierarchy structure. Each structure in this field represents a group contained in the root group. The following example shows the contents of the second structure in this field.

```
level2 = toplevel.Groups(2)

level2 = 

        Filename: 'C:\matlab\toolbox\matlab\demos\example.h5'
            Name: '/g2'
          Groups: []
        Datasets: [1x2 struct]
       Datatypes: []
           Links: []
      Attributes: []
```

In the sample file, the group named /g2 contains two data sets. The following figure illustrates this part of the sample HDF5 file organization.



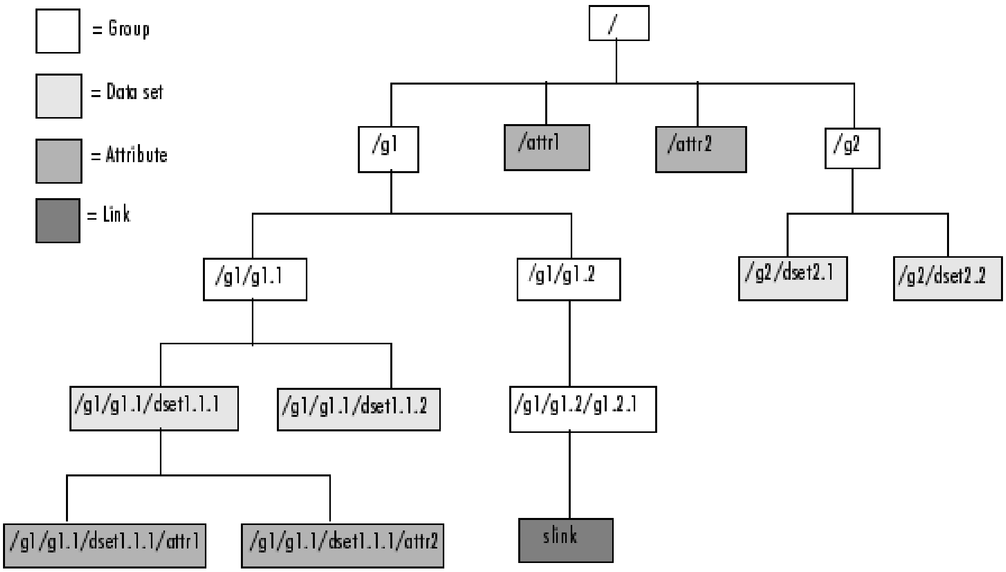
Organization of the Data Set /g2 in the Sample HDF5 File

To get information about a data set, look at either of the structures returned in the Datasets field. These structures provide information about the data set, such as its name, dimensions, and data type.

```
dataset1 = level2.Datasets(1)

dataset1 =
      Filename: 'L:\matlab\toolbox\matlab\demos\example.h5'
          Name: '/g2/dset2.1'
          Rank: 1
      Datatype: [1x1 struct]
          Dims: 10
       MaxDims: 10
        Layout: 'contiguous'
    Attributes: []
         Links: []
     Chunksize: []
     Fillvalue: []
```

By examining the structures at each level of the hierarchy, you can traverse the entire file. The following figure describes the complete hierarchical organization of the sample file `example.h5`.



Hierarchical Structure of example.h5 HDF5 File

Importing Data from an HDF5 File

To read data or metadata from an HDF5 file, use the `hdf5read` function. As arguments, you must specify the name of the HDF5 file and the name of the data set or attribute. Alternatively, you can specify just the field in the structure returned by `hdf5info` that contains the name of the data set or attribute; `hdf5read` can determine the file name from the `Filename` field in the structure. For more information about finding the name of a data set or attribute in an HDF5 file, see “Determining the Contents of an HDF5 File” on page 7-20.

To illustrate, this example reads the data set, `/g2/dset2.1` from the HDF5 sample file `example.h5`.

```
data = hdf5read('example.h5','/g2/dset2.1');
```

The return value contains the values in the data set, in this case a 1-by-10 vector of single-precision values:

```
data =

    1.0000
    1.1000
    1.2000
    1.3000
    1.4000
    1.5000
    1.6000
    1.7000
    1.8000
    1.9000
```

The `hdf5read` function maps HDF5 data types to appropriate MATLAB data types, whenever possible. If the HDF5 file contains data types that cannot be represented in MATLAB, `hdf5write` uses one of the predefined MATLAB HDF5 data type objects to represent the data.

For example, if an HDF5 data set contains four array elements, `hdf5read` can return the data as a 1-by-4 array of `hdf5.h5array` objects:

```
whos

Name    Size         Bytes     Class

data     1x4                 hdf5.h5array

Grand total is 4 elements using 0 bytes
```

For more information about the MATLAB HDF5 data type objects, see “Mapping HDF5 Data Types to MATLAB Data Types” on page 7-27.

Exporting Data to HDF5 Files

To write data or metadata from the MATLAB workspace to an HDF5 file, use the `hdf5write` function. As arguments, specify:

- Name of an existing HDF5 file, or the name you want to assign to a new file.

- Name of an existing data set or attribute, or the name you want to assign to a new data set or attribute. To learn how to determine the name of data sets in an existing HDF5 file, see "Determining the Contents of an HDF5 File" on page 7-20.
- Data or metadata you want to write to the file. `hdf5write` converts MATLAB data types to the appropriate HDF5 data type automatically. For nonatomic data types, you can also create HDF5 objects to represent the data.

This example creates a 5-by-5 array of `uint8` values and then writes the array to an HDF5 file. By default, `hdf5write` overwrites the file, if it already exists. The example specifies an `hdf5write` mode option to append data to existing file.

1 Create a MATLAB variable in the workspace. This example creates a 5-by-5 array of `uint8` values.

```
testdata = uint8(magic(5))
```

2 Write the data to an HDF5 file. As arguments to `hdf5read`, the example specifies the name you want to assign to the HDF5 file, the name you want to assign to the data set, and the MATLAB variable.

```
hdf5write('myfile.h5', '/dataset1', testdata)
```

To add data to an existing file, you must use the `'writemode'` option, specifying the`'append'` value. The file must already exist and it cannot already contain a data set with the same name

```
hdf5write('myfile.h5', '/dataset12', testdata,'writemode','append')
```

If you are writing simple data sets, such as scalars, strings, or a simple compound data types, you can just pass the data directly to `hdf5write`; this function automatically maps the MATLAB data types to appropriate HDF5 data types. However, if your data is a complex data set, you might need to use one of the predefined MATLAB HDF5 objects to pass the data to the `hdf5write` function. The HDF5 objects are designed for situations where the mapping between MATLAB and HDF5 types is ambiguous.

For example, when passed a cell array of strings, the `hdf5write` function writes a data set made up of strings, not a data set of arrays containing

strings. If that is not the mapping you intend, use HDF5 objects to specify the correct mapping. In addition, note that HDF5 makes a distinction between the size of a data set and the size of a data type. In MATLAB, data types are always scalar. In HDF5, data types can have a size; that is, types can be either scalar (like MATLAB) or m-by-n. In HDF5, a 5-by-5 data set containing a single `uint8` value in each element is distinct from a 1-by-1 data set containing a 5-by-5 array of `uint8` values. In the first case, the data set contains 25 observations of a single value; in the second case, the data set contains a single observation with 25 values. For more information about the MATLAB HDF5 data type objects, see "Mapping HDF5 Data Types to MATLAB Data Types" on page 7-27.

Mapping HDF5 Data Types to MATLAB Data Types

When the `hdf5read` function reads data from an HDF5 file into the MATLAB workspace, it maps HDF5 data types to MATLAB data types, depending on whether the data in the data set is in an *atomic* data type or a nonatomic *composite* data type.

Mapping Atomic Data Types. Atomic data types describe commonly used binary formats for numbers (integers and floating point) and characters (ASCII). Because different computing architectures and programming languages support different number and character representations, the HDF5 library provides the platform-independent data types, which it then maps to an appropriate data type for each platform. For example, a computer may support 8-, 16-, 32-, and 64-bit signed integers, stored in memory in little endian byte order.

If the data in the data set is stored in one of the HDF5 atomic data types, `hdf5read` uses the equivalent MATLAB data type to represent the data. Each data set contains a `Datatype` field that names the data type. For example, the data set `/g2/dset2.2` in the sample HDF5 file includes atomic data and data type information.

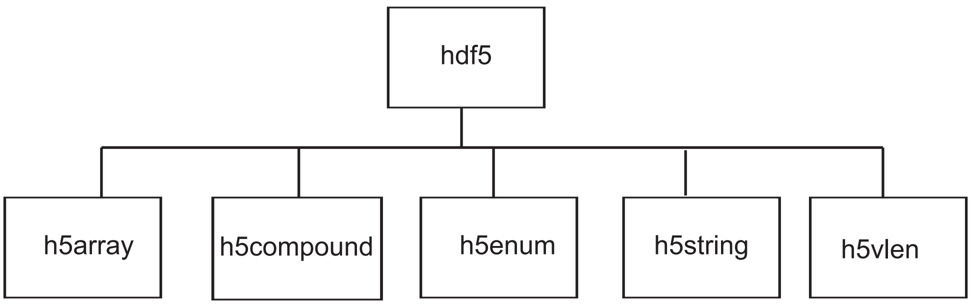
```
dtype = dataset1.Datatype
dtype =

        Name: []
       Class: 'H5T_IEEE_F32BE'
    Elements: []
```

The H5T_IEEE_F32BE class name indicates the data is a 4-byte, big endian, IEEE floating-point data type. (See the HDF5 specification for more information about atomic data types.)

Mapping Composite Data Types. A composite data type is an aggregation of one or more atomic data types. Composite data types include structures, multidimensional arrays, and variable-length data types (one-dimensional arrays).

If the data in the data set is stored in one of the HDF5 nonatomic data types and the data cannot be represented in the workspace using a native MATLAB data type,hdf5read uses one of a set of classes MATLAB defines to represent HDF5 data types. The following figure illustrates the hdf5 class and its subclasses. For more information about a specific class, see the sections that follow. To learn more about the HDF5 data types in general, see the HDF Web page at http://www.hdfgroup.org.



For example, if an HDF5 file contains a data set made up of an enumerated data type which cannot be represented in MATLAB, hdf5read uses the HDF5 h5enum class to represent the data. An h5enum object has data members that store the enumerations (text strings), their corresponding values, and the enumerated data.

You might also need to use these HDF5 data type classes when using the hdf5write function to write data from the MATLAB workspace to an HDF5 file. By default, hdf5write can convert most MATLAB data to appropriate HDF5 data types. However, if this default data type mapping is not suitable, you can create HDF5 data types directly.

To access the data in the data set in the MATLAB workspace, you must access the `Data` field in the object.

This example converts a simple MATLAB vector into an `h5array` object and then displays the fields in the object:

```
vec = [ 1 2 3];

hhh = hdf5.h5array(vec);

hhh:

    Name: ''
    Data: [1 2 3]

hhh.Data

ans =

     1     2     3
```

MATLAB HDF5 h5array Data Class. The `h5array` data class associates a name with an array. The following tables list the class constructors, data members, and methods.

Constructors	Description
`arr = hdf5.h5array`	Creates an `h5array` object.
`arr = hdf5.h5array(data)`	Creates an `h5array` object, where `data` specifies the value of the `Data` member. `data` can be numeric, a cell array, or an HDF5 data type.

Data Members	Description
`Data`	Multidimensional array
`Name`	Text string specifying name of the object

Methods	Description
`setData(arr, data)`	Sets the value of the `Data` member, where `arr` is an `h5array` object and `data` can be numeric, a cell array, or an HDF5 data type.
`setName(arr, name)`	Sets the value of the `Name` member, where `arr` is an `h5array` object and `name` is a string or cell array.

MATLAB HDF5 h5compound Data Class. The `h5compound` data class associates a name with a structure. You can define the field names in the structure and their values. The following tables list the class constructors, data members, and methods.

Constructors	Description
`C = hdf5.h5compound`	Creates an `h5compound` object.
`C = hdf5.h5compound(n1,n2,...)`	Creates an `h5compound` object, where `n1`, `n2` and so on are text strings that specify field names. The constructor creates a corresponding data field for every member name.

Data Members	Description
`Data`	Multidimensional array
`Name`	Text string specifying name of the object
`MemberNames`	Text strings specifying name of the object

Methods	Description
`addMember(C, mName)`	Creates a new field in the object `C`. `mName` specifies the name of the field.
`setMember(C, mName, mData)`	Sets the value of the `Data` element associated with the field specified by `mName`, where `C` is an `h5compound` object and `mData` can be numeric or an HDF5 data type.

Methods	Description
`setMemberNames(C, n1, n2,...)`	Specifies the names of fields in the structure, where `C` is an `h5compound` object and `n1`,`n2`, and so on are text strings that specify field names. The method creates a corresponding data field for every name specified.
`setName(C, name)`	Sets the value of the `Name` member, where `C` is an `h5compound` object and `name` is a string or cell array.

MATLAB HDF5 h5enum Data Class. The `h5enum` data class defines an enumerated type. You can specify the enumerations (text strings) and the values they represent. The following tables list the class constructors, data members, and methods.

Constructors	Description
`E = hdf5.h5enum`	Creates an `h5enum` object.
`E = hdf5.h5enum(eNames, eVals)`	Creates an `h5enum` object, where `eNames` is a cell array of strings, and `eVals` is vector of integers. `eNames` and `eVals` must have the same number of elements.

Data Members	Description
`Data`	Multidimensional array
`Name`	Text string specifying name of the object
`EnumNames`	Text string specifying the enumerations, that is, the text strings that represent values
`EnumValues`	Values associated with enumerations

Methods	Description
`defineEnum(E, eNames, eVals)`	Defines the set of enumerations with the integer values they represent where `eNames` is a cell array of strings, and `eVals` is vector of integers. `eNames` and `eVals` must have the same number of elements.

Methods	Description
`enumdata = getString(E)`	Returns a cell array containing the names of the enumerations, where `E` is an `h5enum` object.
`setData(E, eData)`	Sets the value of the object's `Data` member, where `E` is an `h5enum` object and `eData` is a vector of integers.
`setEnumNames(E, eNames)`	Specifies the enumerations, where `E` is an `h5enum` object and `eNames` is a cell array of strings.
`setEnumValues(E, eVals)`	Specifies the value associated with each enumeration, where `E` is an `h5enum` object and `eVals` is a vector of integers.
`setName(E, name)`	Sets the value of the object's `Name` member, where `E` is an `h5enum` object and `name` is a string or cell array.

This example uses an HDF5 enumeration object.

1 Create an HDF5 enumerated object.

```
enum_obj = hdf5.h5enum;
```

2 Define the enumerated values and their corresponding names.

```
enum_obj.defineEnum({'RED' 'GREEN' 'BLUE'}, uint8([1 2 3]));
```

`enum_obj` now contains the definition of the enumeration that associates the names `RED`, `GREEN`, and `BLUE` with the numbers 1, 2, and 3.

3 Add enumerated data to the object.

```
enum_obj.setData(uint8([2 1 3 3 2 3 2 1]));
```

In the HDF5 file, these numeric values map to the enumerated values `GREEN`, `RED`, `BLUE`, `BLUE`, `GREEN`, etc.

4 Write the enumerated data to a data set named `objects` in an HDF5 file.

```
hdf5write('myfile3.h5', '/g1/objects', enum_obj);
```

5 Read the enumerated data set from the file.

```
ddd = hdf5read('myfile3.h5','/g1/objects')

hdf5.h5enum:

          Name: ''
          Data: [2 1 3 3 2 3 2 1]
     EnumNames: {'RED'  'GREEN'  'BLUE'}
    EnumValues: [1 2 3]
```

MATLAB HDF5 h5string Data Class. The `h5string` data class associates a name with a text string and provides optional padding behavior. The following tables list the class constructors, data members, and methods.

Constructors	Description
`str = hdf5.h5string`	Creates an `h5string` object.
`str = hdf5.h5string(data)`	Creates an `h5string` object, where `data` is a text string.
`str = hdf5.h5string(data, padtype)`	Creates an `h5string`object, where `data` is a text string and `padtype` specifies the type of padding to use.

Data Members	Description
`Data`	Multidimensional array
`Name`	Text string specifying name of the object
`Length`	Scalar defining length of string

`Padding`	Type of padding to use:
	`'spacepad'`
	`'nullterm'`
	`'nullpad'`

Methods	Description
`setData(str, data)`	Sets the value of the object's `Data` member, where `str` is an `h5string` object and`data` is a text string.

Methods	Description
`setLength(str, lenVal)`	Sets the value of the object's `Length` member, where `str` is an `h5string` object and `lenVal` is a scalar.
`setName(str, name)`	Sets the value of the object's `Name` member, where `str` is an `h5string` object and `name` is a string or cell array.
`setPadding(str, padType)`	Specifies the value of the object's `Padding` member, where `str` is an `h5string` object and `padType` is a text string specifying one of the supported pad types.

This example uses an HDF5 string object.

1 Create an HDF5 string object, specifying the text string you want it to contain.

```
myH5str = hdf5.h5string('this is a string')

hdf5.h5string:

      Name: ''
    Length: 16
   Padding: 'nullterm'
      Data: 'this is a string'
```

2 See how the generated object is of class `hdf5.h5string` in the workspace.

```
whos
  Name         Size     Bytes   Class           Attributes

  myH5str      1x1              hdf5.h5string
```

3 Set the name of the object, using a HDF5 string object method, and view the object again.

```
setName( myH5str, 'my H5 string object')

myH5str
```

```
hdf5.h5string:

       Name: 'my H5 string object'
     Length: 16
    Padding: 'nullterm'
       Data: 'this is a string'
```

MATLAB HDF5 h5vlen Data Class. The `h5vlen` data class creates a variable-length array, that is, an array in which the elements can have different lengths. This is also called a ragged array. The following tables list the class constructors, data members, and methods.

Constructors	Description
`V = hdf5.h5vlen`	Creates an `h5vlen` object.
`V = hdf5.h5vlen(data)`	Creates an `h5vlen` object, where `data` specifies the value of the `Data` member. `data` can be numeric or an HDF5 data type.

Data Members	Description
`Data`	Multidimensional array
`Name`	Text string specifying name of the object

Methods	Description
`setData(V, data)`	Sets the value of the object's `Data` member, where `V` is an `h5vlen` object and `data` can be a scalar, vector, text string, a cell array, or an HDF5 data type.
`setName(V, name)`	Sets the value of the object's`Name` member, where `V` is an `h5vlen` object and `name` is a string or cell array.

The following example creates an array of HDF5 `h5vlen` objects. The `h5vlen` objects contain numeric vectors of various lengths.

```
v(1) = hdf5.h5vlen([1:5]);
```

```
v(2) = hdf5.h5vlen([7:-1:3]);
v(3) = hdf5.h5vlen([1:2:8]);
```

Using the MATLAB Low-Level HDF5 Functions

MATLAB provides direct access to the over 200 functions in the HDF5 library by creating MATLAB functions that correspond to the functions in the HDF5 library. In this way, you can access the features of the HDF5 library from MATLAB, such as reading and writing complex data types and using the HDF5 subsetting capabilities.

The HDF5 library organizes the library functions into groups, called *interfaces*. For example, all the routines related to working with files, such as opening and closing, are in the H5F interface, where *F* stands for file. MATLAB organizes the low-level HDF5 functions into classes that correspond to each HDF5 interface. For example, the MATLAB functions that correspond to the HDF5 file interface (H5F) are in the @H5F class directory. For a complete list of the HDF5 interfaces and the corresponding MATLAB class directories, see `hdf5`.

The following sections provide more details about how to use the MATLAB HDF5 low-level functions. Topics covered include:

- "Mapping HDF5 Function Syntax to MATLAB Function Syntax" on page 7-37
- "Mapping Between HDF5 Data Types and MATLAB Data Types" on page 7-39
- "Example: Using the MATLAB HDF5 Low-level Functions" on page 7-41
- "Preserving the Correct Layout of Your Data" on page 7-44

Note This section does not attempt to describe all features of the HDF5 library or explain basic HDF5 programming concepts. To use the MATLAB HDF5 low-level functions effectively, you must refer to the official HDF5 documentation available at the HDF Web site (`http://www.hdfgroup.org`).

Mapping HDF5 Function Syntax to MATLAB Function Syntax

In most cases, the syntax of the MATLAB low-level HDF5 functions is identical to the syntax of the corresponding HDF5 library functions. For example, the following is the function signature of the `H5Fopen` function in the HDF5 library. In the HDF5 function signatures, `hid_t` and `herr_t` are HDF5 types that return numeric values that represent object identifiers or error status values.

```
hid_t H5Fopen(const char *name, unsigned flags, hid_t access_id ) /* C syntax */
```

In MATLAB, each function in an HDF5 interface is a method of a MATLAB class. To view the function signature for a function, specify the class directory name and then the function name, as in the following.

```
help @H5F/open
```

The following shows the signature of the corresponding MATLAB function. First note that, because it's a method of a class, you must use the dot notation to call the MATLAB function: `H5F.open`. This MATLAB function accepts the same three arguments as the HDF5 function: a text string for the name, an HDF5-defined constant for the flags argument, and an HDF5 property list ID. You use property lists to specify characteristics of many different HDF5 objects. In this case, it's a file access property list. Refer to the HDF5 documentation to see which constants can be used with a particular function and note that, in MATLAB, constants are passed as text strings.

```
file_id = H5F.open(name, flags, plist_id)
```

There are, however, some functions where the MATLAB function signature is different than the corresponding HDF5 library function. The following sections describe some general differences that you should keep in mind when using the MATLAB low-level HDF5 functions.

- "Output Parameters Become Return Values" on page 7-38
- "String Length Parameters Unnecessary" on page 7-38
- "Use Empty Array to Specify NULL" on page 7-38
- "Specifying Multiple Constants" on page 7-39

Output Parameters Become Return Values. Some HDF5 library functions use function parameters to return data on the right-hand side (RHS) of the function signature, i.e. as input parameters. The corresponding MATLAB function, because MATLAB allows multiple return values, moves these output parameters to the left-hand side (LHS) of the function signature, i.e. as return values. To illustrate, look at the `H5Dread` function. This function returns data in the `buf` parameter.

```
herr_t H5Dread(hid_t dataset_id, hid_t mem_type_id, hid_t mem_space_id,
        hid_t file_space_id, hid_t xfer_plist_id, void * buf ) /* C syntax */
```

The corresponding MATLAB function changes the output parameter `buf` into a return value. Note that the HDF5 error return is not used. In MATLAB, the nonzero or negative value `herr_t` return values become MATLAB errors. Use MATLAB `try-catch` statements to handle errors.

```
buf = H5D.read(dataset_id, mem_type_id, mem_space_id, file_space_id, plist_id)
```

String Length Parameters Unnecessary. The length parameter used by some HDF5 library functions to specify the length of string parameters are not necessary in the corresponding MATLAB function. For example, the `H5Aget_name` function in the HDF5 library includes a buffer as an output parameter and the size of the buffer as an input parameter.

```
ssize_t H5Aget_name(hid_t attr_id,size_t buf_size,char *buf ) /* C syntax */
```

The corresponding MATLAB function changes the output parameter `buf` into a return value and drops the `buf_size` parameter:

```
attr_name = H5A.get_name(attr_id)
```

Use Empty Array to Specify NULL. The MATLAB functions use empty arrays (`[]`) where HDF5 library functions accept the value `NULL`. For example, the `H5Dfill` function in the HDF5 library accepts the value `NULL` in place of a specified fill value.

```
herr_t H5Dfill(const void *fill, hid_t fill_type_id, void *buf,
               hid_t buf_type_id, hid_t space_id ) /* C syntax */
```

When using the corresponding MATLAB function, you can specify an empty array (`[]`) instead of `NULL`.

Specifying Multiple Constants. Some functions in the HDF5 library require you to specify an array of constants. For example, in the `H5Screate_simple` function, if you want to specify that each dimension in the data space can be unlimited, you use the constant `H5S_UNLIMITED` for each dimension in `maxdims`. In MATLAB, because you pass constants as text strings, you must use a cell array to achieve the same result. The following code fragment provides an example of using a cell array to specify this constant for each dimension of this data space.

```
ds_id = H5S.create_simple(2,[3 4],{'H5S_UNLIMITED' 'H5S_UNLIMITED'});
```

Mapping Between HDF5 Data Types and MATLAB Data Types

When the HDF5 low-level functions read data from an HDF5 file or write data to an HDF5 file, the functions map HDF5 data types to MATLAB data types automatically.

For *atomic* data types, such as commonly used binary formats for numbers (integers and floating point) and characters (ASCII), the mapping is typically straightforward because MATLAB supports similar types. See the table Mapping Between HDF5 Atomic Data Types and MATLAB® Data Types on page 7-39 for a list of these mappings.

Mapping Between HDF5 Atomic Data Types and MATLAB Data Types

HDF5 Atomic Data Type	MATLAB Data Type
Bit-field	Array of packed 8-bit integers
Float	MATLAB single and double types, provided that they occupy 64 bits or fewer
Integer types, signed and unsigned	Equivalent MATLAB integer types, signed and unsigned
Opaque	Array of `uint8` values
Reference	Array of `uint8` values
String	MATLAB character arrays.

For *composite* data types, such as aggregations of one or more atomic data types into structures, multidimensional arrays, and variable-length data types (one-dimensional arrays), the mapping is sometimes ambiguous with reference to the HDF5 data type. In HDF5, a 5-by-5 data set containing a single `uint8` value in each element is distinct from a 1-by-1 data set containing a 5-by-5 array of `uint8` values. In the first case, the data set contains 25 observations of a single value; in the second case, the data set contains a single observation with 25 values. In MATLAB both of these data sets are represented by a 5-by-5 matrix.

If your data is a complex data set, you might need to create HDF5 data types directly to make sure you have the mapping you intend. See the table Mapping Between HDF5 Composite Data Types and MATLAB® Data Types on page 7-40 for a list of the default mappings. You can specify the data type when you write data to the file using the `H5Dwrite` function. See the HDF5 data type interface documentation for more information.

Mapping Between HDF5 Composite Data Types and MATLAB Data Types

HDF5 Composite Data Type	MATLAB Data Type
Array	Extends the dimensionality of the data type which it contains. For example, an array of an array of integers in HDF5 would map onto a two dimensional array of integers in MATLAB.
Compound	MATLAB structure. Note: All structures representing HDF5 data in MATLAB are scalar.
Enumeration	Array of integers which each have an associated name
Variable Length	MATLAB 1-D cell arrays

Reporting Data Set Dimensions. The MATLAB low-level HDF5 functions report data set dimensions and the shape of data sets differently than the MATLAB high-level functions. For ease of use, the MATLAB high-level functions report data set dimensions consistent with MATLAB column-major indexing. To be consistent with the HDF5 library, and to support the possibility of nested data sets and complicated data types, the MATLAB low-level functions report array dimensions using the C row-major orientation.

Example: Using the MATLAB HDF5 Low-level Functions

This example shows how to use the MATLAB HDF5 low-level functions to write a data set to an HDF5 file and then read the data set from the file.

1. Create the MATLAB variable that you want to write to the HDF5 file. The examples creates a two-dimensional array of `uint8` data.

   ```
   testdata = [1 3 5; 2 4 6];
   ```

2. Create the HDF5 file or open an existing file. The example creates a new HDF5 file, named `my_file.h5`, in the system temp directory.

   ```
   filename = fullfile(tempdir,'my_file.h5');

   fileID = H5F.create(filename,'H5F_ACC_TRUNC','H5P_DEFAULT','H5P_DEFAULT');
   ```

   In HDF5, you use the `H5Fcreate` function to create a file. The example uses the MATLAB equivalent, `H5F.create`. As arguments, specify the name you want to assign to the file, the type of access you want to the file (`'H5F_ACC_TRUNC'` in the example), and optional additional characteristics specified by a file creation property list and a file access property list. This example uses default values for these property lists (`'H5P_DEFAULT'`). In the example, note how the C constants are passed to the MATLAB functions as strings. The function returns an ID to the HDF5 file.

3. Create the data set in the file to hold the MATLAB variable. In the HDF5 programming model, you must define the data type and dimensionality (data space) of the data set as separate entities.

   a. Specify the data type used by the data set. In HDF5, you use the `H5Tcopy` function to create integer or floating-point data types. The example uses the corresponding MATLAB function, `H5T.copy`, to create a `double` data type because the MATLAB data is `double`. The function returns a data type ID.

      ```
      datatypeID = H5T.copy('H5T_NATIVE_DOUBLE');
      ```

   b. Specify the dimensions of the data set. In HDF5, you use the `H5Screate_simple` routine to create a data space. The example uses the corresponding MATLAB function, `H5S.create_simple`, to specify the dimensions. The function returns a data space ID.

Because HDF5 stores data in row-major order and the MATLAB array is organized in column-major order, you should reverse the ordering of the dimension extents before using H5Screate_simple to preserve the layout of the data. You can use fliplr for this purpose. For a list of other HDF5 functions that require dimension flipping, see "Preserving the Correct Layout of Your Data" on page 7-44.

```
dims = size(testdata);
dataspaceID = H5S.create_simple(3, fliplr(dims), []);
```

Other software programs that use row-major ordering (such as H5DUMP from the HDF Group) may report the size of the dataset to be 3-by-2 instead of 2-by-3.

c Create the data set. In HDF5, you use the H5Dcreate routine to create a data set. The example uses the corresponding MATLAB function, H5D.create, specifying the file ID, the name you want to assign to the data set, data type ID, the data space ID, and a data set creation property list ID as arguments. The example uses the defaults for the property lists. The function returns a data set ID.

```
dsetname = 'my_dataset';
datasetID = H5D.create(fileID,dsetname,datatypeID,dataspaceID,'H5P_DEFAULT');
```

Note To write a large data set, you must use the chunking capability of the HDF5 library. To do this, create a property list and use the H5P.set_chunk function to set the chunk size in the property list. In the following example, the dimensions of the data set are dims = [2^16 2^16] and the chunk size is 1024-by-1024. You then pass the property list as the last argument to the data set creation function, H5D.create, instead of using the H5P_DEFAULT value.

```
plistID = H5P.create('H5P_DATASET_CREATE'); % create property list

chunk_size = min([1024 1024], dims); % define chunk size
H5P.set_chunk(plistID, chunk_size); % set chunk size in property list

datasetID = H5D.create(fileID, dsetname, datatypeID, dataspaceID, plistID);
```

---

4 Write the data to the data set. In HDF5, you use the `H5Dwrite` routine to write data to a data set. The example uses the corresponding MATLAB function, `H5D.write`, specifying as arguments the data set ID, the memory data type ID, the memory space ID, the data space ID, the transfer property list ID and the name of the MATLAB variable to be written to the data set.

You can use the memory data type to specify the data type used to represent the data in the file. The example uses the constant `'H5ML_DEFAULT'` which lets the MATLAB function do an automatic mapping to HDF5 data types. The memory data space ID and the data set's data space ID specify to write subsets of the data set to the file. The example uses the constant `'H5S_ALL'` to write all the data to the file and uses the default property list.

```
H5D.write(datasetID,'H5ML_DEFAULT','H5S_ALL','H5S_ALL', ...
          'H5P_DEFAULT', testdata);
```

If you had not reversed the ordering of the dimension extents in step 3b above, you would have been required to permute the MATLAB array before using `H5D.write`, which could result in an enormous performance penalty.

5 Close the data set, data space, data type, and file objects. If used inside a MATLAB function, these identifiers are closed automatically when they go out of scope.

```
H5D.close(datasetID);
H5S.close(dataspaceID);
H5T.close(datatypeID);
H5F.close(fileID);
```

6 To read the data set you wrote to the file, you must open the file. In HDF5, you use the `H5Fopen` routine to open an HDF5 file, specifying the name of the file, the access mode, and a property list as arguments. The example uses the corresponding MATLAB function, `H5F.open`, opening the file for read-only access.

```
fileID = H5F.open(filename,'H5F_ACC_RDONLY','H5P_DEFAULT');
```

7 After opening the file, you must open the data set. In HDF5, you use the `H5Dopen` function to open a data set. The example uses the corresponding MATLAB function, `H5D.open`, specifying as arguments the file ID and the name of the data set, defined earlier in the example.

```
datasetID = H5D.open(fileID, dsetname);
```

8 After opening the data set, you can read the data into the MATLAB workspace. In HDF5, you use the `H5Dread` function to read an HDF5 file. The example uses the corresponding MATLAB function, `H5D.read`, specifying as arguments the data set ID, the memory data type ID, the memory space ID, the data space ID, and the transfer property list ID.

```
returned_data = H5D.read(datasetID,'H5ML_DEFAULT',...
                         'H5S_ALL','H5S_ALL','H5P_DEFAULT');
```

You can compare the original MATLAB variable, `testdata`, with the variable just created, `data`, to see if they are the same.

Preserving the Correct Layout of Your Data

When you use any of the following functions that deal with dataspaces, you should flip dimension extents to preserve the correct layout of the data, as illustrated in step 3b in “Example: Using the MATLAB HDF5 Low-level Functions” on page 7-41.

- `H5D.set_extent`
- `H5P.get_chunk`
- `H5P.set_chunk`
- `H5S.create_simple`
- `H5S.get_simple_extent_dims`
- `H5S.select_hyperslab`
- `H5T.array_create`
- `H5T.get_array_dims`

Hierarchical Data Format (HDF4) Files

In this section...

Note For information about importing HDF5 data, which is a separate, incompatible format, see "Hierarchical Data Format (HDF5) Files" on page 7-20.

Using the HDF Import Tool

Hierarchical Data Format (HDF4) is a general-purpose, machine-independent standard for storing scientific data in files, developed by the National Center for Supercomputing Applications (NCSA). For more information about these file formats, read the HDF documentation at the HDF Web site (`www.hdfgroup.org`).

HDF-EOS is an extension of HDF4 that was developed by the National Aeronautics and Space Administration (NASA) for storage of data returned from the Earth Observing System (EOS). For more information about this extension to HDF4, see the HDF-EOS documentation at the NASA Web site (`www.hdfeos.org`).

The HDF Import Tool is a graphical user interface that you can use to navigate through HDF4 or HDF-EOS files and import data from them. Importing data using the HDF Import Tool involves these steps:

- "Step 5: Closing HDF Files and the HDF Import Tool" on page 7-50

The following sections provide more detail about each of these steps.

Step 1: Opening an HDF4 File in the HDF Import Tool

Open an HDF4 or HDF-EOS file in MATLAB using one of the following methods:

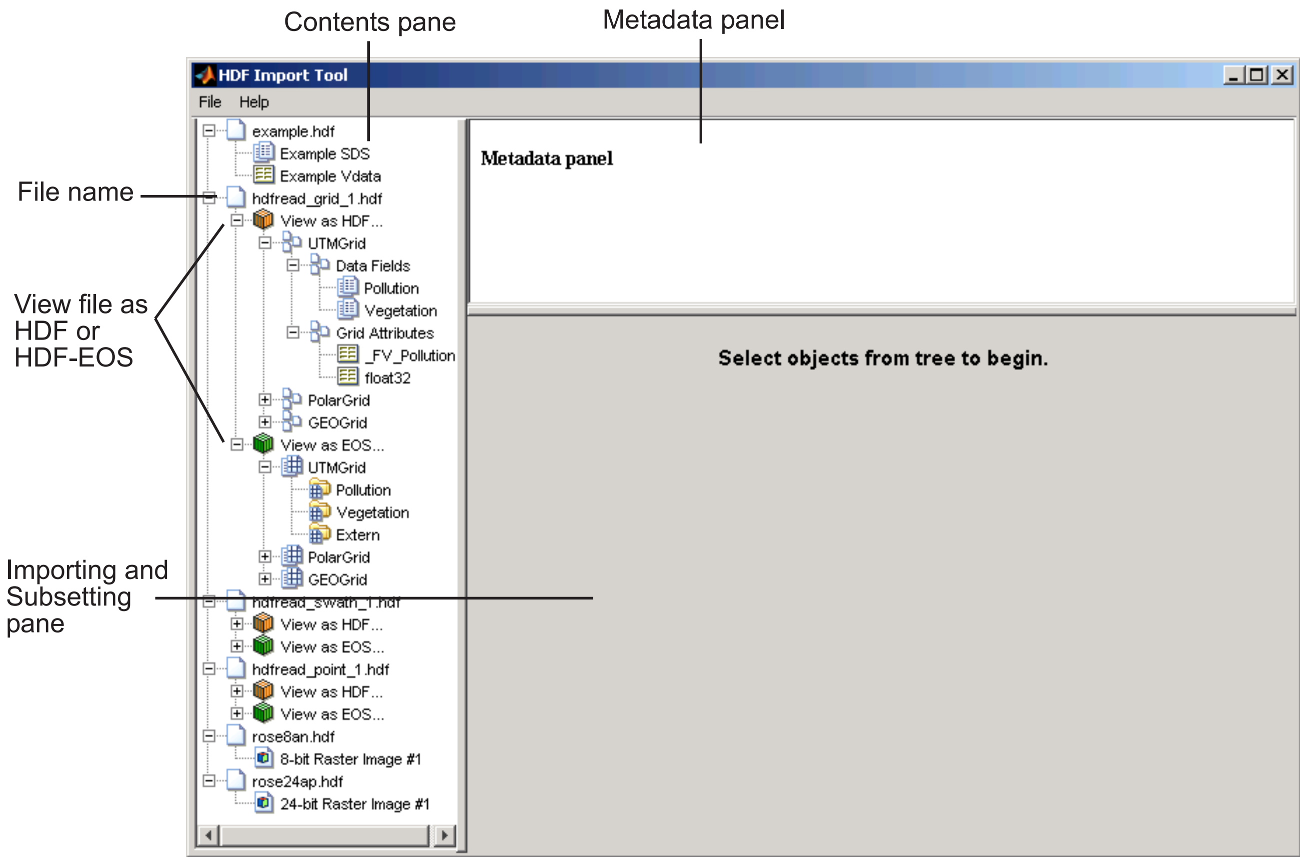
- Choose the **Import Data** option from the MATLAB **File** menu. If you select an HDF4 or HDF-EOS file, the MATLAB Import Wizard automatically starts the HDF Import Tool.
- Start the HDF Import Tool by entering the `hdftool` command at the MATLAB command line:

  ```
  hdftool
  ```

  This opens an empty HDF Import Tool. To open a file, click the **Open** option on the HDFTool **File** menu and select the file you want to open. You can open multiple files in the HDF Import Tool.
- Open an HDF or HDF-EOS file by specifying the file name with the `hdftool` command on the MATLAB command line:

  ```
  hdftool('example.hdf')
  ```

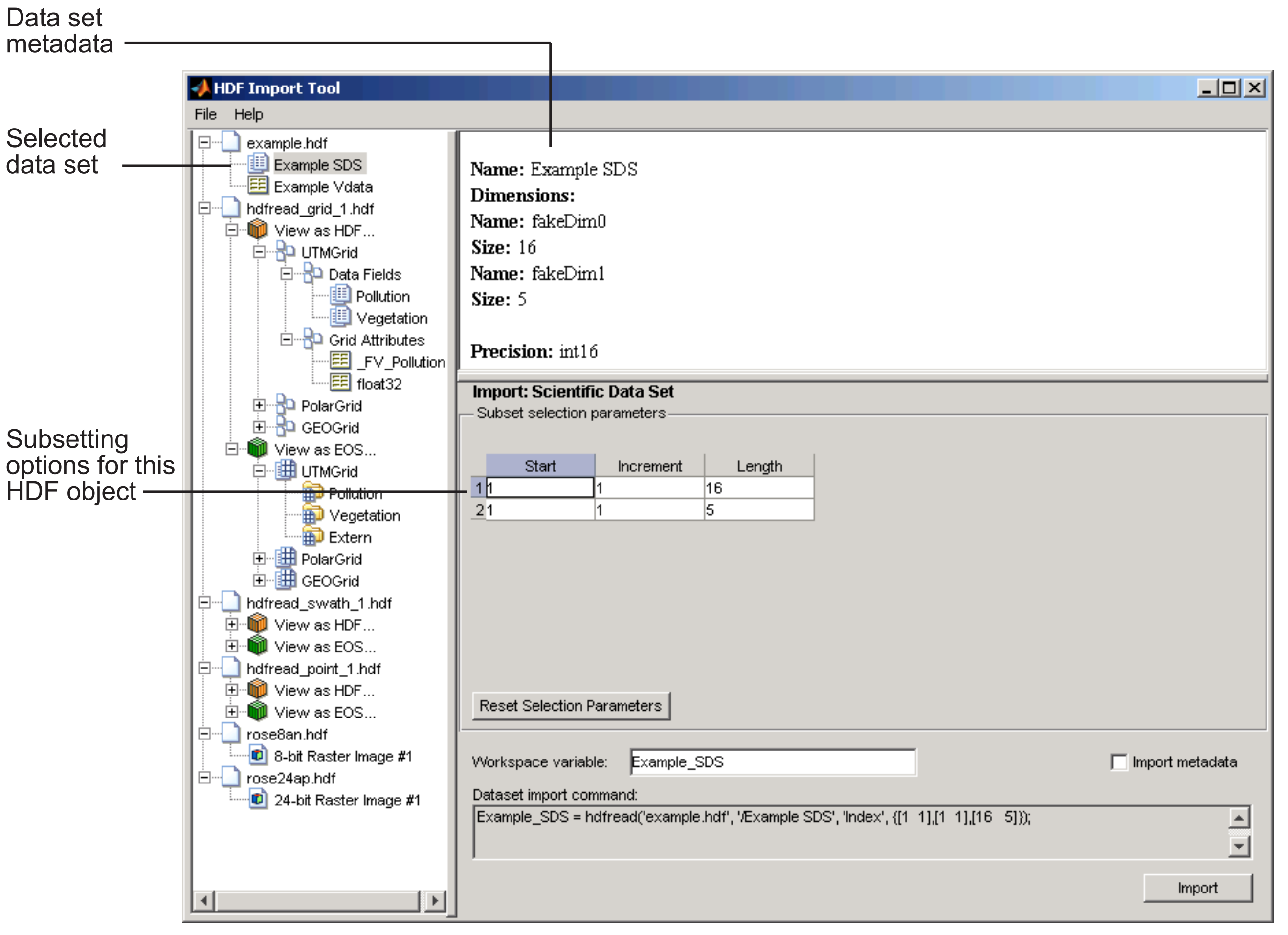
Viewing a File in the HDF Import Tool. When you open an HDF4 or HDF-EOS file in the HDF Import Tool, the tool displays the contents of the file in the Contents pane. You can use this pane to navigate within the file to see what data sets it contains. You can view the contents of HDF-EOS files as HDF data sets or as HDF-EOS files. The icon in the contents pane indicates the view, as illustrated in the following figure. Note that these are just two views of the same data.



Step 2: Selecting a Data Set in an HDF File

To import a data set, you must first select the data set in the contents pane of the HDF Import Tool. Use the Contents pane to view the contents of the file and navigate to the data set you want to import.

For example, the following figure shows the data set `Example SDS` in the HDF file selected. Once you select a data set, the Metadata panel displays information about the data set and the importing and subsetting pane displays subsetting options available for this type of HDF object.



Step 3: Specifying a Subset of the Data (Optional)

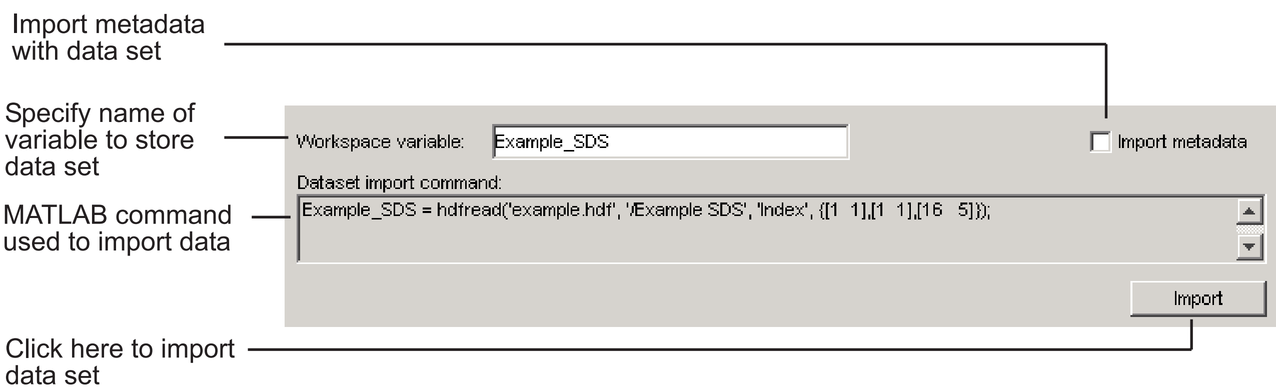
When you select a data set in the contents pane, the importing and subsetting pane displays the subsetting options available for that type of HDF object. The subsetting options displayed vary depending on the type of HDF object. For more information, see "Using the HDF Import Tool Subsetting Options" on page 7-50.

Step 4: Importing Data and Metadata

To import the data set you have selected, click the **Import** button, bottom right corner of the Importing and Subsetting pane. Using the Importing and Subsetting pane, you can

- Specify the name of the workspace variable — By default, the HDF Import Tool uses the name of the HDF4 data set as the name of the MATLAB workspace variable. In the following figure, the variable name is `Example_SDS`. To specify another name, enter text in the **Workspace Variable** text box.
- Specify whether to import metadata associated with the data set — To import any metadata that might be associated with the data set, select the **Import Metadata** check box. To store the metadata, the HDF Import Tool creates a second variable in the workspace with the same name with "`_info`" appended to it. For example, if you select this check box, the name of the metadata variable for the data set in the figure would be `Example_SDS_info`.
- Save the data set import command syntax — The **Dataset import command** text window displays the MATLAB command used to import the data set. This text is not editable, but you can copy and paste it into the MATLAB Command Window or a text editor for reuse.

The following figure shows how to specify these options in the HDF Import Tool.



Step 5: Closing HDF Files and the HDF Import Tool

To close a file, select the file in the contents pane and click **Close File** on the HDF Import Tool **File** menu.

To close all the files open in the HDF Import Tool, click **Close All Files** on the HDF Import Tool **File** menu.

To close the tool, click **Close HDFTool** in the HDF Import Tool **File** menu or click the **Close** button in the upper right corner of the tool.

If you used the `hdftool` syntax that returns a handle to the tool,

```
h = hdftool('example.hdf')
```

you can use the `close(h)` command to close the tool from the MATLAB command line.

Using the HDF Import Tool Subsetting Options

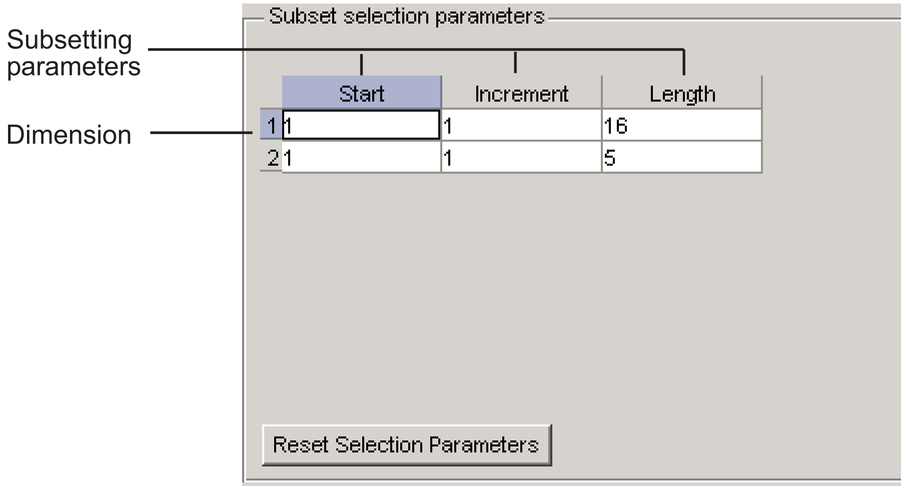
When you select a data set, the importing and subsetting pane displays the subsetting options available for that type of data set. The following sections provide information about these subsetting options for all supported data set types. For general information about the HDF Import tool, see "Using the HDF Import Tool" on page 7-45.

- "HDF Scientific Data Sets (SD)" on page 7-51
- "HDF Vdata" on page 7-51
- "HDF-EOS Grid Data" on page 7-53
- "HDF-EOS Point Data" on page 7-58
- "HDF-EOS Swath Data" on page 7-58
- "HDF Raster Image Data" on page 7-62

Note To use these data subsetting options effectively, you must understand the HDF and HDF-EOS data formats. Therefore, use this documentation in conjunction with the HDF documentation (`www.hdfgroup.org`) and the HDF-EOS documentation (`www.hdfeos.org`).

HDF Scientific Data Sets (SD)

The HDF scientific data set (SD) is a group of data structures used to store and describe multidimensional arrays of scientific data. Using the HDF Import Tool subsetting parameters, you can import a subset of an HDF scientific data set by specifying the location, range, and number of values to be read along each dimension.



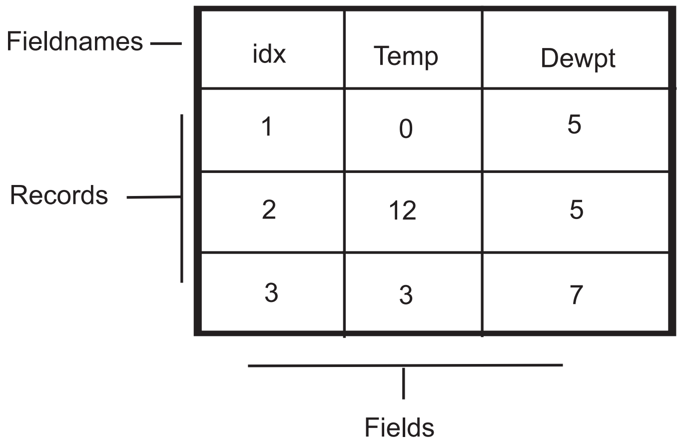
The subsetting parameters are:

- **Start** — Specifies the position on the dimension to begin reading. The default value is 1, which starts reading at the first element of each dimension. The values specified must not exceed the size of the relevant dimension of the data set.
- **Increment** — Specifies the interval between the values to read. The default value is 1, which reads every element of the data set.
- **Length** — Specifies how much data to read along each dimension. The default value is the length of the dimension, which causes all the data to be read.

HDF Vdata

HDF Vdata data sets provide a framework for storing customized tables. A Vdata table consists of a collection of records whose values are stored in

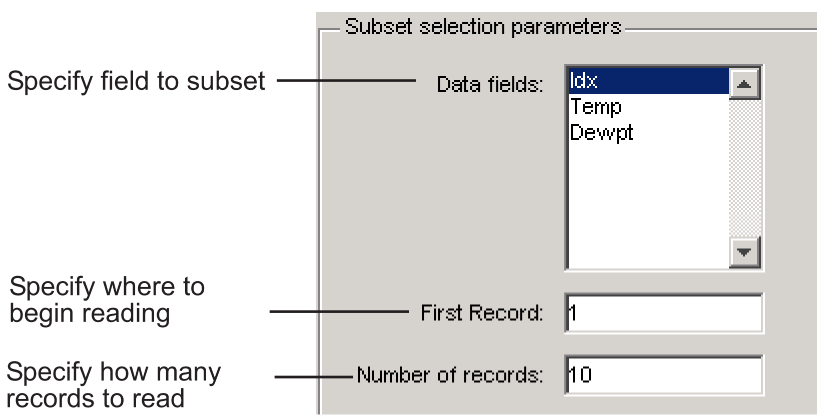
fixed-length fields. All records have the same structure and all values in each field have the same data type. Each field is identified by a name. The following figure illustrates a Vdata table.



You can import a subset of an HDF Vdata data set in the following ways:

- Specifying the name of the field that you want to import
- Specifying the range of records that you want to import

The following figure shows how you specify these subsetting parameters for Vdata.

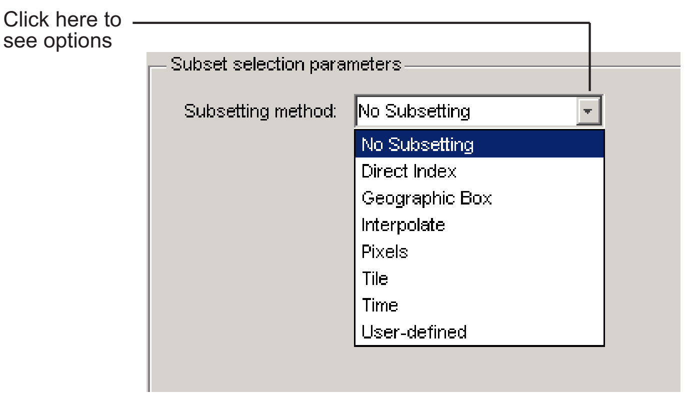


HDF-EOS Grid Data

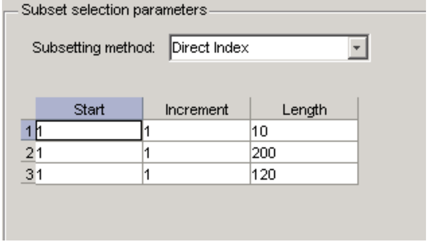
In HDF-EOS Grid data, a rectilinear grid overlays a map. The map uses a known map projection. The HDF Import Tool supports the following mutually exclusive subsetting options for Grid data:

- “Direct Index” on page 7-53
- “Geographic Box” on page 7-54
- “Interpolation” on page 7-55
- “Pixels” on page 7-56
- “Tile” on page 7-56
- “Time” on page 7-56
- “User-Defined” on page 7-57

To access these options, click the Subsetting method menu in the importing and subsetting pane.



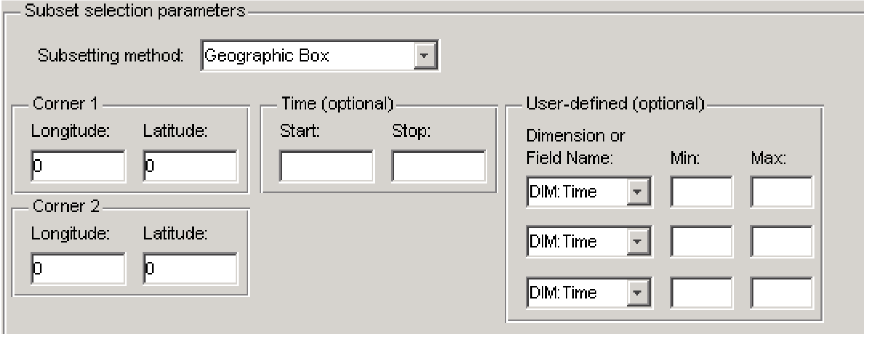
Direct Index. You can import a subset of an HDF-EOS Grid data set by specifying the location, range, and number of values to be read along each dimension.



Each row represents a dimension in the data set and each column represents these subsetting parameters:

- **Start** — Specifies the position on the dimension to begin reading. The default value is 1, which starts reading at the first element of each dimension. The values specified must not exceed the size of the relevant dimension of the data set.
- **Increment** — Specifies the interval between the values to read. The default value is 1, which reads every element of the data set.
- **Length** — Specifies how much data to read along each dimension. The default value is the length of the dimension, which causes all the data to be read.

Geographic Box. You can import a subset of an HDF-EOS Grid data set by specifying the rectangular area of the grid that you are interested in. To define this rectangular area, you must specify two points, using longitude and latitude in decimal degrees. These points are two corners of the rectangular area. Typically, **Corner 1** is the upper-left corner of the box, and **Corner 2** is the lower-right corner of the box.

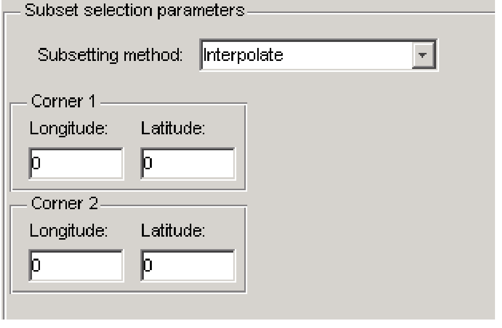


Optionally, you can further define the subset of data you are interested in by using Time parameters (see “Time” on page 7-56) or by specifying other User-Defined subsetting parameters (see “User-Defined” on page 7-57).

Interpolation. Interpolation is the process of estimating a pixel value at a location in between other pixels. In interpolation, the value of a particular pixel is determined by computing the weighted average of some set of pixels in the vicinity of the pixel.

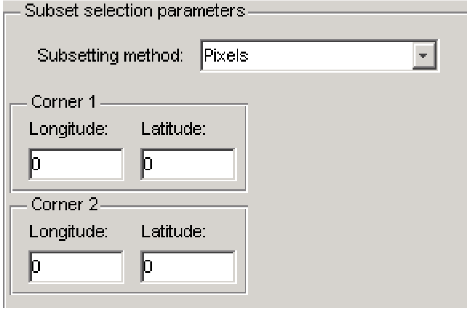
You define the region used for bilinear interpolation by specifying two points that are corners of the interpolation area:

- **Corner 1** – Specify longitude and latitude values in decimal degrees. Typically, **Corner 1** is the upper-left corner of the box.
- **Corner 2** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 2** is the lower-right corner of the box

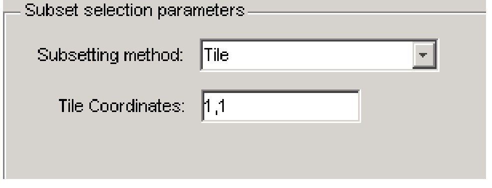


Pixels. You can import a subset of the pixels in a Grid data set by defining a rectangular area over the grid. You define the region used for bilinear interpolation by specifying two points that are corners of the interpolation area:

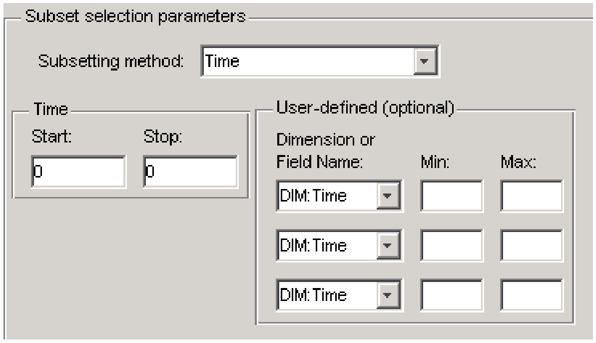
- **Corner 1** – Specify longitude and latitude values in decimal degrees. Typically, **Corner 1** is the upper-left corner of the box.
- **Corner 2** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 2** is the lower-right corner of the box



Tile. In HDF-EOS Grid data, a rectilinear grid overlays a map. Each rectangle defined by the horizontal and vertical lines of the grid is referred to as a *tile*. If the HDF-EOS Grid data is stored as tiles, you can import a subset of the data by specifying the coordinates of the tile you are interested in. Tile coordinates are 1-based, with the upper-left corner of a two-dimensional data set identified as `1,1`. In a three-dimensional data set, this tile would be referenced as `1,1,1`.

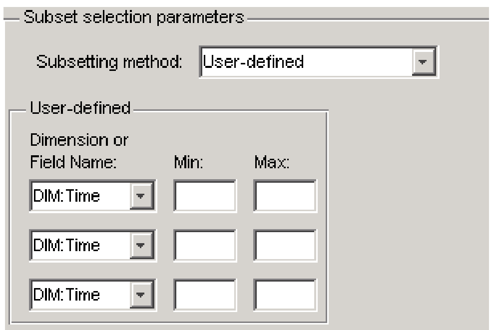


Time. You can import a subset of the Grid data set by specifying a time period. You must specify both the start time and the stop time (the endpoint of the time span). The units (hours, minutes, seconds) used to specify the time are defined by the data set.



Along with these time parameters, you can optionally further define the subset of data to import by supplying user-defined parameters.

User-Defined. You can import a subset of the Grid data set by specifying user-defined subsetting parameters.

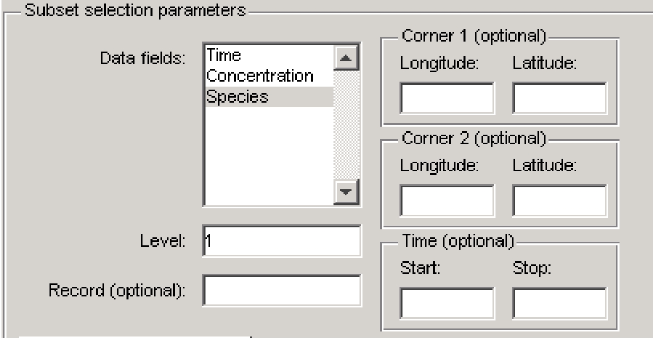


When specifying user-defined parameters, you must first specify whether you are subsetting along a dimension or by field. Select the dimension or field by name using the **Dimension or Field Name** menu. Dimension names are prefixed with the characters `DIM:`.

Once you specify the dimension or field, you use **Min** and **Max** to specify the range of values that you want to import. For dimensions, **Min** and **Max** represent a range of *elements*. For fields, **Min** and **Max** represent a range of *values*.

HDF-EOS Point Data

HDF-EOS Point data sets are tables. You can import a subset of an HDF-EOS Point data set by specifying field names and level. Optionally, you can refine the subsetting by specifying the range of records you want to import, by defining a rectangular area, or by specifying a time period. For information about specifying a rectangular area, see "Geographic Box" on page 7-54. For information about subsetting by time, see "Time" on page 7-56.



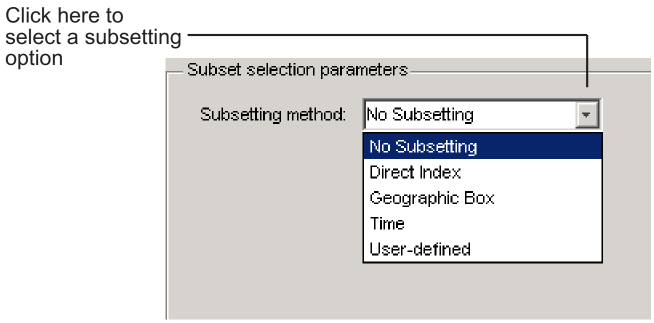
HDF-EOS Swath Data

HDF-EOS Swath data is data that is produced by a satellite as it traces a path over the earth. This path is called its ground track. The sensor aboard the satellite takes a series of scans perpendicular to the ground track. Swath data can also include a vertical measure as a third dimension. For example, this vertical dimension can represent the height above the Earth of the sensor.

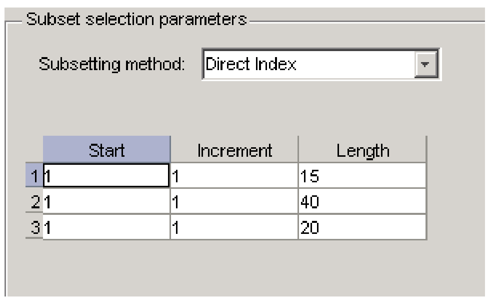
The HDF Import Tool supports the following mutually exclusive subsetting options for Swath data:

- "Direct Index" on page 7-59
- "Geographic Box" on page 7-60
- "Time" on page 7-61
- "User-Defined" on page 7-61

To access these options, click the `Subsetting method` menu in the **Importing and Subsetting** pane.



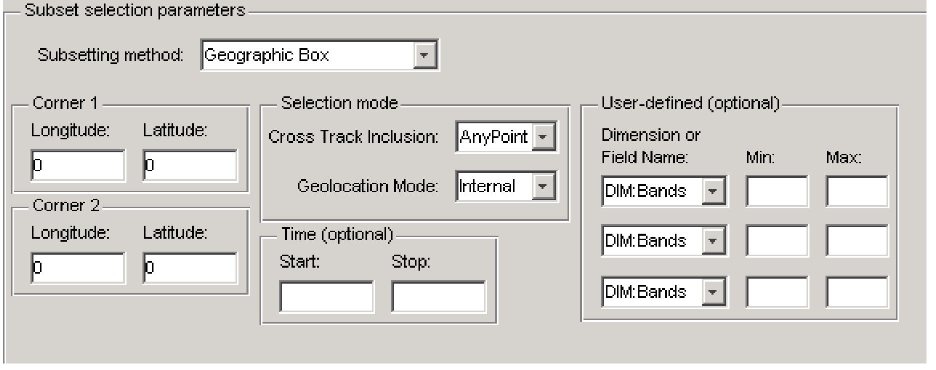
Direct Index. You can import a subset of an HDF-EOS Swath data set by specifying the location, range, and number of values to be read along each dimension.



Each row represents a dimension in the data set and each column represents these subsetting parameters:

- **Start** — Specifies the position on the dimension to begin reading. The default value is 1, which starts reading at the first element of each dimension. The values specified must not exceed the size of the relevant dimension of the data set.
- **Increment** — Specifies the interval between the values to read. The default value is 1, which reads every element of the data set.
- **Length** — Specifies how much data to read along each dimension. The default value is the length of the dimension, which causes all the data to be read.

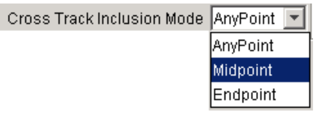
Geographic Box. You can import a subset of an HDF-EOS Swath data set by specifying the rectangular area of the grid that you are interested in and by specifying the selection Mode.



You define the rectangular area by specifying two points that specify two corners of the box:

- **Corner 1** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 1** is the upper-left corner of the box.
- **Corner 2** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 2** is the lower-right corner of the box.

You specify the selection mode by choosing the type of **Cross Track Inclusion** and the **Geolocation mode**. The **Cross Track Inclusion** value determines how much of the area of the geographic box that you define must fall within the boundaries of the swath.

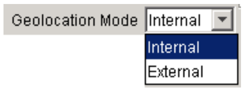


Select from these values:

- **AnyPoint** — Any part of the box overlaps with the swath.
- **Midpoint** — At least half of the box overlaps with the swath.

- **Endpoint** — All of the area defined by the box overlaps with the swath.

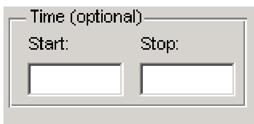
The **Geolocation Mode** value specifies whether geolocation fields and data must be in the same swath.



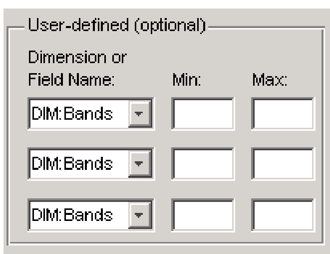
Select from these values:

- **Internal** — Geolocation fields and data fields must be in the same swath.
- **External** — Geolocation fields and data fields can be in different swaths.

Time. You can optionally also subset swath data by specifying a time period. The units used (hours, minutes, seconds) to specify the time are defined by the data set



User-Defined. You can optionally also subset a swath data set by specifying user-defined parameters.



When specifying user-defined parameters, you must first specify whether you are subsetting along a dimension or by field. Select the dimension or field by name using the **Dimension or Field Name** menu. Dimension names are prefixed with the characters `DIM:`.

Once you specify the dimension or field, you use **Min** and **Max** to specify the range of values that you want to import. For dimensions, **Min** and **Max** represent a range of *elements*. For fields, **Min** and **Max** represent a range of *values*.

HDF Raster Image Data

For 8-bit HDF raster image data, you can specify the colormap.

Using the MATLAB HDF4 High-Level Functions

To import data from an HDF or HDF-EOS file, you can use the MATLAB HDF4 high-level function `hdfread`. The `hdfread` function provides a programmatic way to import data from an HDF4 or HDF-EOS file that still hides many of the details that you need to know if you use the low-level HDF functions, described in “Using the HDF4 Low-Level Functions” on page 7-65. You can also import HDF4 data using an interactive GUI, described in “Using the HDF Import Tool” on page 7-45.

This section describes these high-level MATLAB HDF functions, including

- “Using hdfinfo to Get Information About an HDF4 File” on page 7-62
- “Using hdfread to Import Data from an HDF4 File” on page 7-63

To export data to an HDF4 file, you must use the MATLAB HDF4 low-level functions.

Using hdfinfo to Get Information About an HDF4 File

To get information about the contents of an HDF4 file, use the `hdfinfo` function. The `hdfinfo` function returns a structure that contains information about the file and the data in the file.

Note You can also use the HDF Import Tool to get information about the contents of an HDF4 file. See “Using the HDF Import Tool” on page 7-45 for more information.

This example returns information about a sample HDF4 file included with MATLAB:

```
info = hdfinfo('example.hdf')

info =

    Filename: 'example.hdf'
         SDS: [1x1 struct]
       Vdata: [1x1 struct]
```

To get information about the data sets stored in the file, look at the SDS field.

Using hdfread to Import Data from an HDF4 File

To use thehdfread function, you must specify the data set that you want to read. You can specify the filename and the data set name as arguments, or you can specify a structure returned by the hdfinfo function that contains this information. The following example shows both methods. For information about how to import a subset of the data in a data set, see "Reading a Subset of the Data in a Data Set" on page 7-65.

1 Determine the names of data sets in the HDF4 file, using the hdfinfo function.

   ```
   info = hdfinfo('example.hdf')

   info =

       Filename: 'example.hdf'
            SDS: [1x1 struct]
          Vdata: [1x1 struct]
   ```

   To determine the names and other information about the data sets in the file, look at the contents of the SDS field. The Name field in the SDS structure gives the name of the data set.

   ```
   dsets = info.SDS

   dsets =
   ```

```
     Filename: 'example.hdf'
         Type: 'Scientific Data Set'
         Name: 'Example SDS'
         Rank: 2
     DataType: 'int16'
   Attributes: []
         Dims: [2x1 struct]
        Label: {}
  Description: {}
        Index: 0
```

2 Read the data set from the HDF4 file, using the `hdfread` function. Specify the name of the data set as a parameter to the function. Note that the data set name is case sensitive. This example returns a 16-by-5 array:

```
dset = hdfread('example.hdf', 'Example SDS');

dset =

     3     4     5     6     7
     4     5     6     7     8
     5     6     7     8     9
     6     7     8     9    10
     7     8     9    10    11
     8     9    10    11    12
     9    10    11    12    13
    10    11    12    13    14
    11    12    13    14    15
    12    13    14    15    16
    13    14    15    16    17
    14    15    16    17    18
    15    16    17    18    19
    16    17    18    19    20
    17    18    19    20    21
    18    19    20    21    22
```

Alternatively, you can specify the specific field in the structure returned by `hdfinfo` that contains this information. For example, to read a scientific data set, use the `SDS` field.

```
dset = hdfread(info.SDS);
```

Reading a Subset of the Data in a Data Set. To read a subset of a data set, you can use the optional `'index'` parameter. The value of the index parameter is a cell array of three vectors that specify the location in the data set to start reading, the skip interval (e.g., read every other data item), and the amount of data to read (e.g., the length along each dimension). In HDF4 terminology, these parameters are called the *start*, *stride*, and *edge* values.

For example, this code

- Starts reading data at the third row, third column (`[3 3]`).
- Reads every element in the array (`[]`).
- Reads 10 rows and 2 columns (`[10 2]`).

```
subset = hdfread('Example.hdf','Example SDS',...
                   'Index',{[3 3],[],[10 2 ]})

subset =

     7     8
     8     9
     9    10
    10    11
    11    12
    12    13
    13    14
    14    15
    15    16
    16    17
```

Using the HDF4 Low-Level Functions

This section describes how to use MATLAB functions to access the HDF4 Application Programming Interfaces (APIs). These APIs are libraries of C routines that you can use to import data from an HDF4 file or export data from the MATLAB workspace into an HDF4 file. To import or export data, you must use the functions in the HDF4 API associated with the particular HDF4 data type you are working with. Each API has a particular programming model, that is, a prescribed way to use the routines to write data sets to the file. To illustrate this concept, this section describes the programming

model of one particular HDF4 API: the HDF4 Scientific Data (SD) API. For a complete list of the HDF4 APIs supported by MATLAB and the functions you use to access each one, see the `hdf` reference page.

Note This section does not attempt to describe all HDF4 features and routines. To use the MATLAB HDF4 functions effectively, you must refer to the official NCSA documentation at the HDF Web site (`www.hdfgroup.org`).

Topics covered include

- "Understanding the HDF4 to MATLAB Syntax Mapping" on page 7-66
- "Example: Importing Data Using the HDF4 SD API Functions" on page 7-67
- "Example: Exporting Data Using the HDF4 SD API Functions" on page 7-73
- "Using the MATLAB HDF4 Utility API" on page 7-80

Understanding the HDF4 to MATLAB Syntax Mapping

Each HDF4 API includes many individual routines that you use to read data from files, write data to files, and perform other related functions. For example, the HDF4 Scientific Data (SD) API includes separate C routines to open (`SDopen`), close (`SDend`), and read data (`SDreaddata`).

Instead of supporting each routine in the HDF4 APIs, MATLAB provides a single function that serves as a gateway to all the routines in a particular HDF4 API. For example, the HDF Scientific Data (SD) API includes the C routine `SDend` to close an HDF4 file:

```
status = SDend(sd_id); /* C code */
```

To call this routine from MATLAB, use the MATLAB function associated with the SD API, `hdfsd`. You must specify the name of the routine, minus the API acronym, as the first argument and pass any other required arguments to the routine in the order they are expected. For example,

```
status = hdfsd('end',sd_id); % MATLAB code
```

Handling HDF4 Routines with Output Arguments. Some HDF4 API routines use output arguments to return data. Because MATLAB does not support output arguments, you must specify these arguments as return values.

For example, the `SDfileinfo` routine returns data about an HDF4 file in two output arguments, `ndatasets` and `nglobal_atts`. Here is the C code:

```
status = SDfileinfo(sd_id, ndatasets, nglobal_atts);
```

To call this routine from MATLAB, change the output arguments into return values:

```
[ndatasets, nglobal_atts, status] = hdfsd('fileinfo',sd_id);
```

Specify the return values in the same order as they appear as output arguments. The function status return value is always specified as the last return value.

Example: Importing Data Using the HDF4 SD API Functions

To illustrate using HDF4 API routines in MATLAB, the following sections provide a step-by-step example of how to import HDF4 Scientific Data (SD) into the MATLAB workspace.

- "Step 1: Opening the HDF4 File" on page 7-68
- "Step 2: Retrieving Information About the HDF4 File" on page 7-68
- "Step 3: Retrieving Attributes from an HDF4 File (Optional)" on page 7-69
- "Step 4: Selecting the Data Sets to Import" on page 7-70
- "Step 5: Getting Information About a Data Set" on page 7-70
- "Step 6: Reading Data from the HDF4 File" on page 7-71
- "Step 7: Closing the HDF4 Data Set" on page 7-72
- "Step 8: Closing the HDF4 File" on page 7-73

Note The following sections, when referring to specific routines in the HDF4 SD API, use the C library name rather than the MATLAB function name. The MATLAB syntax is used in all examples.

Step 1: Opening the HDF4 File. To import an HDF4 SD data set, you must first open the file using the SD API routine `SDstart`. (In HDF4 terminology, the numeric arrays stored in HDF4 files are called data sets.) In MATLAB, you use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `start` in this case.
- Name of the file you want to open.
- Mode in which you want to open it. The following table lists the file access modes supported by the `SDstart` routine. In MATLAB, you specify these modes as text strings. You can specify the full HDF4 mode name or one of the abbreviated forms listed in the table.

HDF4 File Creation Mode	HDF4 Mode Name	MATLAB String
Create a new file	`'DFACC_CREATE'`	`'create'`
Read access	`'DFACC_RDONLY'`	`'read'` or `'rdonly'`
Read and write access	`'DFACC_RDWR'`	`'rdwr'` or `'write'`

For example, this code opens the file `mydata.hdf` for read access:

```
sd_id = hdfsd('start','mydata.hdf','read');
```

If `SDstart` can find and open the file specified, it returns an HDF4 SD file identifier, named `sd_id` in the example. Otherwise, it returns `-1`.

Step 2: Retrieving Information About the HDF4 File. To get information about an HDF4 file, you must use the SD API routine `SDfileinfo`. This function returns the number of data sets in the file and the number of global attributes in the file, if any. (For more information about global attributes, see "Example: Exporting Data Using the HDF4 SD API Functions" on page 7-73.) In MATLAB, you use the `hdfsd` function, specifying the following arguments:

- Name of the SD API routine, `fileinfo` in this case
- SD file identifier, `sd_id`, returned by `SDstart`

In this example, the HDF4 file contains three data sets and one global attribute.

```
[ndatasets, nglobal_atts, stat] = hdfsd('fileinfo',sd_id)

ndatasets =
    3

nglobal_atts =
    1

status =
    0
```

Step 3: Retrieving Attributes from an HDF4 File (Optional). HDF4 files can optionally include information, called *attributes*, that describes the data the file contains. Attributes associated with an entire HDF4 file are called *global* attributes. Attributes associated with a data set are called *local* attributes. (You can also associate attributes with files or dimensions. For more information, see "Step 4: Writing Metadata to an HDF4 File" on page 7-78.)

To retrieve attributes from an HDF4 file, use the HDF4 API routine `SDreadattr`. In MATLAB, use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `readattr` in this case.
- File identifier (`sd_id`) returned by `SDstart`, for global attributes, or the data set identifier for local attributes. (See "Step 4: Selecting the Data Sets to Import" on page 7-70 to learn how to get a data set identifier.)
- Index identifying the attribute you want to view. HDF4 uses zero-based indexing. If you know the name of an attribute but not its index, use the `SDfindattr` routine to determine the index value associated with the attribute.

For example, this code returns the contents of the first global attribute, which is the character string `my global attribute`:

```
attr_idx = 0;
[attr, status] = hdfsd('readattr', sd_id, attr_idx);

attr =
    my global attribute
```

Step 4: Selecting the Data Sets to Import. To select a data set, use the SD API routine `SDselect`. In MATLAB, you use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `select` in this case
- HDF4 SD file identifier (`sd_id`) returned by `SDstart`

If `SDselect` finds the specified data set in the file, it returns an HDF4 SD data set identifier, called `sds_id` in the example. If it cannot find the data set, it returns `-1`.

Note Do not confuse HDF4 SD *file* identifiers, named `sd_id` in the examples, with HDF4 SD *data set* identifiers, named `sds_id` in the examples.

```
sds_id = hdfsd('select',sd_id,1)
```

Step 5: Getting Information About a Data Set. To read a data set, you must get information about the data set, such as its name, size, and data type. In the HDF4 SD API, you use the `SDgetinfo` routine to gather this information. In MATLAB, use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `getinfo` in this case
- HDF4 SD data set identifier (`sds_id`) returned by `SDselect`

This code retrieves information about the data set identified by `sds_id`:

```
[dsname, dsndims, dsdims, dstype, dsatts, stat] =
              hdfsd('getinfo',sds_id)
dsname =
      A
```

```
dsndims =
      2

dsdims =
      5     3

dstype =
      double

dsatts =
      0

stat =
      0
```

Step 6: Reading Data from the HDF4 File. To read data from an HDF4 file, you must use the `SDreaddata` routine. In MATLAB, use the `hdfsd` function, specifying as arguments:

- Name of the SD API function, `readdata` in this case.
- HDF4 SD data set identifier (`sds_id`) returned by `SDselect`.
- Location in the data set where you want to start reading data, specified as a vector of index values, called the *start* vector. To read from the beginning of a data set, specify zero for each element of the start vector. Use `SDgetinfo` to determine the dimensions of the data set.
- Number of elements along each dimension to skip between each read operation, specified as a vector of scalar values, called the *stride* vector. To read every element of a data set, specify `1` as the value for each element of the vector or specify an empty array (`[]`).
- Total number of elements to read along each dimension, specified as a vector of scalar values, called the *edges* vector. To read every element of a data set, set each element of the edges vector to the size of each dimension of the data set. Use `SDgetinfo` to determine these sizes.

Note SDgetinfo returns dimension values in row-major order, the ordering used by HDF4. Because MATLAB stores data in column-major order, you must specify the dimensions in column-major order, that is, [columns,rows]. In addition, you must use zero-based indexing in these arguments.

For example, to read the entire contents of a data set, use this code:

```
[ds_name, ds_ndims, ds_dims, ds_type, ds_atts, stat] =
hdfsd('getinfo',sds_id);

ds_start = zeros(1,ds_ndims); % Creates the vector [0 0]
ds_stride = [];
ds_edges = ds_dims;

[ds_data, status] =
            hdfsd('readdata',sds_id,ds_start,ds_stride,ds_edges);

disp(ds_data)
     1     2     3     4     5
     6     7     8     9    10
    11    12    13    14    15
```

To read less than the entire data set, use the start, stride, and edges vectors to specify where you want to start reading data and how much data you want to read. For example, this code reads the entire second row of the sample data set:

```
ds_start = [0 1]; % Start reading at the first column, second row
ds_stride = []; % Read each element
ds_edges = [5 1]; % Read a 1-by-5 vector of data

[ds_data, status] =
           hdfsd('readdata',sds_id,ds_start,ds_stride,ds_edges);
```

Step 7: Closing the HDF4 Data Set. After writing data to a data set in an HDF4 file, you must close access to the data set. In the HDF4 SD API, you use the SDendaccess routine to close a data set. In MATLAB, use the hdfsd function, specifying as arguments:

- Name of the SD API routine, `endaccess` in this case
- HDF4 SD data set identifier (`sds_id`) returned by `SDselect`

For example, this code closes the data set:

```
stat = hdfsd('endaccess',sds_id);
```

You must close access to all the data sets in an HDF4 file before closing it.

Step 8: Closing the HDF4 File. After writing data to a data set and closing the data set, you must also close the HDF4 file. In the HDF4 SD API, you use the `SDend` routine. In MATLAB, use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `end` in this case
- HDF4 SD file identifier (`sd_id`) returned by `SDstart`

For example, this code closes the data set:

```
stat = hdfsd('end',sd_id);
```

Example: Exporting Data Using the HDF4 SD API Functions

The following sections provide a step-by-step example of how to export data from the MATLAB workspace to an HDF4 file using Scientific Data (SD) API functions.

- "Step 1: Creating an HDF4 File" on page 7-74
- "Step 2: Creating an HDF4 Data Set" on page 7-74
- "Step 3: Writing MATLAB Data to an HDF4 File" on page 7-76
- "Step 4: Writing Metadata to an HDF4 File" on page 7-78
- "Step 5: Closing HDF4 Data Sets" on page 7-79
- "Step 6: Closing an HDF4 File" on page 7-80

Step 1: Creating an HDF4 File. To export MATLAB data in HDF4 format, you must first create an HDF4 file, or open an existing one. In the HDF4 SD API, you use the `SDstart` routine. In MATLAB, use the `hdfsd` function, specifying `start` as the first argument. As other arguments, specify

- A text string specifying the name you want to assign to the HDF4 file (or the name of an existing HDF4 file)
- A text string specifying the HDF4 SD interface file access mode

For example, this code creates an HDF4 file named `mydata.hdf`:

```
sd_id = hdfsd('start','mydata.hdf','DFACC_CREATE');
```

When you specify the `DFACC_CREATE` access mode, `SDstart` creates the file and initializes the HDF4 SD multifile interface, returning an HDF4 SD file identifier, named `sd_id` in the example.

If you specify `DFACC_CREATE` mode and the file already exists, `SDstart` fails, returning `-1`. To open an existing HDF4 file, you must use HDF4 read or write modes. For information about using `SDstart` in these modes, see "Step 1: Opening the HDF4 File" on page 7-68.

Step 2: Creating an HDF4 Data Set. After creating the HDF4 file, or opening an existing one, you must create a data set in the file for each MATLAB array you want to export. If you are writing data to an existing data set, you can skip ahead to the next step.

In the HDF4 SD API, you use the `SDcreate` routine to create data sets. In MATLAB, you use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `'create'` in this case
- Valid HDF4 SD file identifier, `sd_id`, returned by `SDstart`
- Name you want assigned to the data set
- Data type of the data set.
- Number of dimensions in the data set. This is called the *rank* of the data set in HDF4 terminology.
- Size of each dimension, specified as a vector

Once you create a data set, you cannot change its name, data type, or dimensions.

For example, to create a data set in which you can write the following MATLAB 3-by-5 array of doubles,

```
A = [ 1 2 3 4 5 ; 6 7 8 9 10 ; 11 12 13 14 15 ];
```

you could call hdfsd, specifying as arguments 'create' and a valid HDF file identifier, sd_id. In addition, set the values of the other arguments as in this code fragment:

```
ds_name = 'A';
ds_type = 'double';
ds_rank = ndims(A);
ds_dims = fliplr(size(A));

sds_id = hdfsd('create',sd_id,ds_name,ds_type,ds_rank,ds_dims);
```

If SDcreate can successfully create the data set, it returns an HDF4 SD data set identifier, (sds_id). Otherwise, SDcreate returns -1.

In this example, note the following:

- The data type you specify in ds_type must match the data type of the MATLAB array that you want to write to the data set. In the example, the array is of class double so the value of ds_type is set to 'double'. If you wanted to use another data type, such as uint8, convert the MATLAB array to use this data type,

  ```
  A = uint8([ 1 2 3 4 5 ; 6 7 8 9 10 ; 11 12 13 14 15 ]);
  ```

  and specify the name of the MATLAB data type, uint8 in this case, in the ds_type argument.

  ```
  ds_type = 'uint8';
  ```

- The code fragment reverses the order of the values in the dimensions argument (ds_dims). This processing is necessary because the MATLAB size function returns the dimensions in column-major order and HDF4 expects to receive dimensions in row-major order.

Step 3: Writing MATLAB Data to an HDF4 File. After creating an HDF4 file and creating a data set in the file, you can write data to the entire data set or just a portion of the data set. In the HDF4 SD API, you use the `SDwritedata` routine. In MATLAB, use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `'writedata'` in this case
- Valid HDF4 SD data set identifier, `sds_id`, returned by `SDcreate`
- Location in the data set where you want to start writing data, called the *start* vector in HDF4 terminology
- Number of elements along each dimension to skip between each write operation, called the *stride* vector in HDF4 terminology
- Total number of elements to write along each dimension, called the *edges* vector in HDF4 terminology
- MATLAB array to be written

Note You must specify the values of the start, stride, and edges arguments in row-major order, rather than the column-major order used in MATLAB. Note how the example uses `fliplr` to reverse the order of the dimensions in the vector returned by the `size` function before assigning it as the value of the edges argument.

The values you assign to these arguments depend on the MATLAB array you want to export. For example, the following code fragment writes this MATLAB 3-by-5 array of doubles,

```
A = [ 1 2 3 4 5; 6 7 8 9 10; 11 12 13 14 15 ];
```

into an HDF4 file:

```
ds_start = zeros(1:ndims(A)); % Start at the beginning
ds_stride = [];               % Write every element.
ds_edges = fliplr(size(A));   % Reverse the dimensions.

stat = hdfsd('writedata',sds_id,...
                         ds_start, ds_stride, ds_edges, A);
```

If it can write the data to the data set, `SDwritedata` returns `0`; otherwise, it returns `-1`.

Note `SDwritedata` queues write operations. To ensure that these queued write operations are executed, you must close the file, using the `SDend` routine. See "Step 6: Closing an HDF4 File" on page 7-80 for more information. As a convenience, MATLAB provides a function, `MLcloseall`, that you can use to close all open data sets and file identifiers with a single call. See "Using the MATLAB HDF4 Utility API" on page 7-80 for more information.

To write less than the entire data set, use the start, stride, and edges vectors to specify where you want to start writing data and how much data you want to write.

For example, the following code fragment uses `SDwritedata` to replace the values of the entire second row of the sample data set:

```
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
```

with the vector `B`:

```
B = [ 9 9 9 9 9];
```

In the example, the start vector specifies that you want to start the write operation in the first column of the second row. Note how HDF4 uses zero-based indexing and specifies the column dimension first. In MATLAB, you would specify this location as `(2,1)`. The edges argument specifies the dimensions of the data to be written. Note that the size of the array of data to be written must match the edge specification.

```
ds_start = [0 1]; % Start writing at the first column, second row.
ds_stride = []; % Write every element.
ds_edges = [5 1]; % Each row is a 1-by-5 vector.

stat = hdfsd('writedata',sds_id,ds_start,ds_stride,ds_edges,B);
```

Step 4: Writing Metadata to an HDF4 File. You can optionally include information in an HDF4 file, called attributes, that describes the file and its contents. Using the HDF4 SD API, you can associate attributes with three types of HDF4 objects:

- An entire HDF4 file — File attributes, also called *global* attributes, generally contain information pertinent to all the data sets in the file.
- A data set in an HDF4 file — Data set attributes, also called *local* attributes, describe individual data sets.
- A dimension of a data set — Dimension attributes provide information about one particular dimension of a data set.

To create an attribute in the HDF4 SD API, use the `SDsetattr` routine. In MATLAB, use the `hdfsd` function, specifying `'setattr'` as the first argument. As other arguments, specify

- A valid HDF4 SD identifier associated with the object. This value can be a file identifier (`sd_id`), a data set identifier (`sds_id`), or a dimension identifier (`dim_id`).
- A text string that defines the name of the attribute.
- The attribute value.

For example, this code creates a global attribute, named `my_global_attr`, and associates it with the HDF4 file identified by `sd_id`:

```
status = hdfsd('setattr',sd_id,'my_global_attr','my_attr_val');
```

Note In the NCSA documentation, the `SDsetattr` routine has two additional arguments: data type and the number of values in the attribute. When calling this routine from MATLAB, you do not have to include these arguments. The MATLAB HDF4 function can determine the data type and size of the attribute from the value you specify.

The SD interface supports predefined attributes that have reserved names and, in some cases, data types. Predefined attributes are identical to user-defined attributes except that the HDF4 SD API has already defined

their names and data types. For example, the HDF4 SD API defines an attribute, named `cordsys`, in which you can specify the coordinate system used by the data set. Possible values of this attribute include the text strings `'cartesian'`, `'polar'`, and `'spherical'`.

Predefined attributes can be useful because they establish conventions that applications can depend on. The HDF4 SD API supports predefined attributes for data sets and dimensions only; there are no predefined attributes for files. For a complete list of the predefined attributes, see the NCSA documentation.

In the HDF4 SD API, you create predefined attributes the same way you create user-defined attributes, using the `SDsetattr` routine. In MATLAB, use the `hdfsd` function, specifying `setattr` as the first argument:

```
attr_name = 'cordsys';
attr_value = 'polar';

status = hdfsd('setattr',sds_id,attr_name,attr_value);
```

The HDF4 SD API also includes specialized functions for writing and reading the predefined attributes. These specialized functions, such as `SDsetdatastrs`, are sometimes easier to use, especially when you are reading or writing multiple related predefined attributes. You must use specialized functions to read or write the predefined dimension attributes.

You can associate multiple attributes with a single HDF4 object. HDF4 maintains an attribute index for each object. The attribute index is zero-based. The first attribute has index value 0, the second has index value 1, and so on. You access an attribute by its index value.

Each attribute has the format `name=value`, where `name` (called `label` in HDF4 terminology) is a text string up to 256 characters in length and `value` contains one or more entries of the same data type. A single attribute can have multiple values.

Step 5: Closing HDF4 Data Sets. After writing data to a data set in an HDF4 file, you must close access to the data set. In the HDF4 SD API, you use the `SDendaccess` routine to close a data set. In MATLAB, use the `hdfsd` function, specifying `endaccess` as the first argument. As the only other argument, specify a valid HDF4 SD data set identifier, `sds_id` in this example:

```
stat = hdfsd('endaccess',sds_id);
```

Step 6: Closing an HDF4 File. After writing data to a data set and closing the data set, you must also close the HDF4 file. In the HDF4 SD API, you use the `SDend` routine. In MATLAB, use the `hdfsd` function, specifying `end` as the first argument. As the only other argument, specify a valid HDF4 SD file identifier, `sd_id` in this example:

```
stat = hdfsd('end',sd_id);
```

You must close access to all the data sets in an HDF4 file before closing it.

Note Closing an HDF4 file executes all the write operations that have been queued using `SDwritedata`. As a convenience, the MATLAB HDF Utility API provides a function that can close all open data set and file identifiers with a single call. See "Using the MATLAB HDF4 Utility API" on page 7-80 for more information.

Using the MATLAB HDF4 Utility API

In addition to the standard HDF4 APIs, listed in the `hdf`reference page, MATLAB supports utility functions that are designed to make it easier to use HDF4 in the MATLAB environment.

For example, using the gateway function to the MATLAB HDF4 utility API, `hdfml`, and specifying the name of the `listinfo` routine as an argument, you can view all the currently open HDF4 identifiers. MATLAB updates this list whenever HDF identifiers are created or closed. In the following example only two identifiers are open.

```
hdfml('listinfo')
No open RI identifiers
No open GR identifiers
No open grid identifiers
No open grid file identifiers
No open annotation identifiers
No open AN identifiers
Open scientific dataset identifiers:
 262144
```

```
Open scientific data file identifiers:
 393216
No open Vdata identifiers
No open Vgroup identifiers
No open Vfile identifiers
No open point identifiers
No open point file identifiers
No open swath identifiers
No open swath file identifiers
No open access identifiers
No open file identifiers
```

Closing All Open HDF4 Identifiers. To close all the currently open HDF4 identifiers in a single call, use the gateway function to the MATLAB HDF4 utility API, `hdfml`, specifying the name of the `closeall` routine as an argument. The following example closes all the currently open HDF4 identifiers.

```
hdfml('closeall')
```

8

Error Handling

Error Reporting in a MATLAB Application

In this section...

Overview

No matter how carefully you plan and test the programs you write, they may not always run as smoothly as expected when executed under different conditions. It is always a good idea to include error checking in programs to ensure reliable operation under all conditions.

In the MATLAB software, you can decide how your programs respond to different types of errors. You may want to prompt the user for more input, display extended error or warning information, or perhaps repeat a calculation using default values. The error-handling capabilities in MATLAB help your programs check for particular error conditions and execute the appropriate code depending on the situation.

When MATLAB detects a severe fault in the command or program it is running, it collects information about what was happening at the time of the error, displays a message to help the user understand what went wrong, and terminates the command or program. This is called *throwing an exception*. You can get an exception while entering commands at the MATLAB command prompt or while executing your program code.

Getting an Exception at the Command Line

If you get an exception at the MATLAB prompt, you have several options on how to deal with it as described below.

Determine the Fault from the Error Message

Evaluate the error message MATLAB has displayed. Most error messages attempt to explain at least the immediate cause of the program failure. There

is often sufficient information to determine the cause and what you need to do to remedy the situation.

Review the Failing Code

If the function in which the error occurred is implemented as an M-file, the error message should include a line that looks something like this:

```
surf

??? Error using ==> surf at 50
Not enough input arguments.
```

The underlined text to the right names the function that threw the error (`surf`, in this case) and shows the failing line number within that function's M-file. Click the underlined text; MATLAB opens the M-file and positions the cursor at the location in the file where the error originated. You may be able to determine the cause of the error by examining this line and the code that precedes it.

Step Through the Code in the Debugger

You can use the MATLAB Debugger to step through the failing code. Click the underlined error text to open the M-file in the MATLAB Editor at or near the point of the error. Next, click the hyphen at the beginning of that line to set a breakpoint at that location. When you rerun your program, MATLAB pauses execution at the breakpoint and enables you to step through the program code. The command `dbstop on error` is also helpful in finding the point of error.

See the documentation on "Editing and Debugging M-Files" for more information.

Getting an Exception in Your Program Code

When you are writing your own program in an M-file, you can *catch* exceptions and attempt to handle or resolve them instead of allowing your program to terminate. When you catch an exception, you interrupt the normal termination process and enter a block of code that deals with the faulty situation. This block of code is called a *catch block*.

Some of the things you might want to do in the catch block are:

- Examine information that has been captured about the error.
- Gather further information to report to the user.
- Try to accomplish the task at hand in some other way.
- Clean up any unwanted side effects of the error.

When you reach the end of the catch block, you can either continue executing the program, if possible, or terminate it.

The documentation on "Capturing Information About the Error" on page 8-5 describes how to acquire information about what caused the error, and "Responding to an Exception" on page 8-17 presents some ideas on how to respond to it.

Generating a New Exception

When your program code detects a condition that will either make the program fail or yield unacceptable results, it should throw an exception. This procedure

- Saves information about what went wrong and what code was executing at the time of the error.
- Gathers any other pertinent information about the error.
- Instructs MATLAB to throw the exception.

The documentation on "Capturing Information About the Error" on page 8-5 describes how to use an `MException` object to capture information about the error, and "Throwing an Exception" on page 8-16 explains how to initiate the exception process.

Capturing Information About the Error

In this section...

Overview

When the MATLAB software throws an exception, it captures information about what caused the error in a data structure called an `MException` object. This object is an instance of the MATLAB `MException` class. You can obtain access to the `MException` object by *catching* the exception before your program aborts and accessing the object constructed for this particular error via the `catch` command. When throwing an exception in response to an error in your own M-file code, you will have to create a new `MException` object and store information about the error in that object.

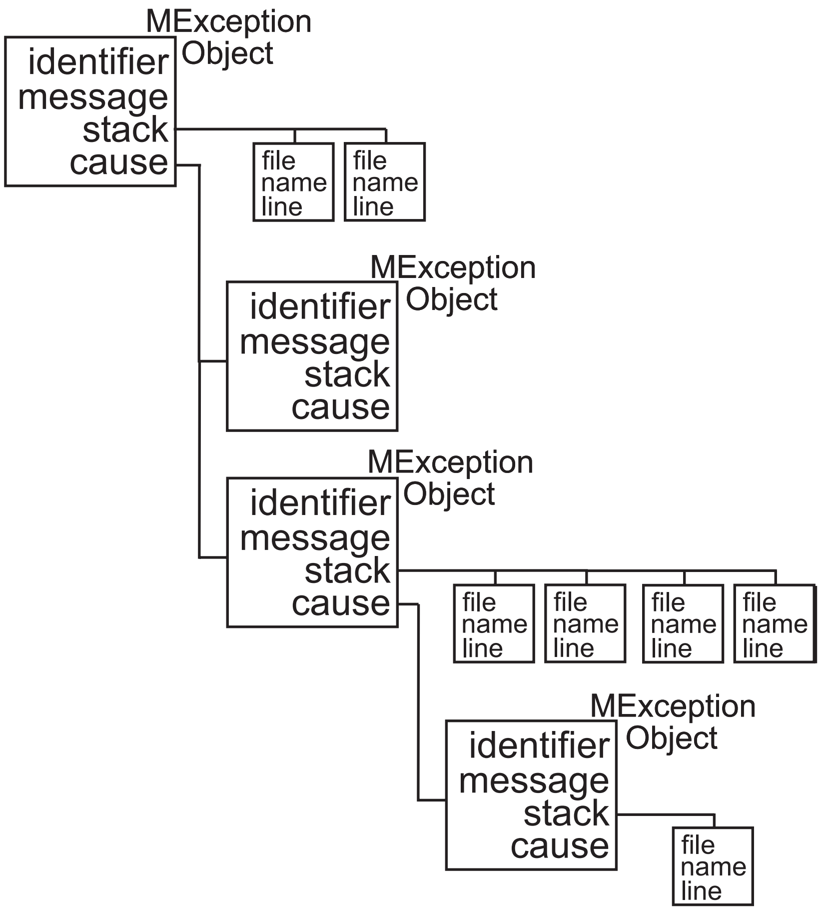
This section describes the `MException` class and objects constructed from that class:

Information on how to use this class is presented in later sections on "Responding to an Exception" on page 8-17 and "Throwing an Exception" on page 8-16.

The MException Class

The figure shown below illustrates one possible configuration of an object of the `MException` class. The object has four properties: `identifier`, `message`, `stack`, and `cause`. Each of these properties is implemented as a field of the structure that represents the `MException` object. The `stack` field is an N-by-1 array of additional structures, each one identifying an M-file, function, and line number from the call stack. The `cause` field is an M-by-1 cell array of `MException` objects, each representing an exception that is related to the current one.

See "Properties of the MException Class" on page 8-7 for a full description of these properties.



Object Constructor

Any code that detects an error and throws an exception must also construct an `MException` object in which to record and transfer information about the error. The syntax of the `MException` constructor is

```
ME = MException(identifier, message)
```

where `identifier` is a MATLAB message identifier of the form

```
component:mnemonic
```

that is enclosed in single quotes, and `message` is a text string, also enclosed in single quotes, that describes the error. The output `ME` is the resulting `MException` object.

If you are responding to an exception rather than throwing one, you do not have to construct an `MException` object. The object has already been constructed and populated by the code that originally detected the error.

Properties of the MException Class

The `MException` class has four properties. Each of these properties is implemented as a field of the structure that represents the `MException` object. Each of these properties is described in the sections below and referenced in the sections on “Responding to an Exception” on page 8-17 and “Throwing an Exception” on page 8-16. All are read-only; their values cannot be changed.

The `MException` properties are:

- `identifier`
- `message`
- `stack`
- `cause`

Repeating the `surf` example shown above, but this time catching the exception, you can see the four properties of the `MException` object structure. (This example uses `try-catch` in an atypical fashion. See the section on “The try-catch Statement” on page 8-17 for more information on using `try-catch`).

```
try
    surf
catch ME
    ME
end
```

Run this at the command line and MATLAB returns the contents of the `MException` object:

```
ME =
 MException object with properties:

    identifier: 'MATLAB:nargchk:notEnoughInputs'
       message: 'Not enough input arguments.'
         stack: [1x1 struct]
         cause: {}
```

The `stack` field shows the filename, function, and line number where the exception was thrown:

```
ME.stack
ans =
    file: 'matlabroot\toolbox\matlab\graph3d\surf.m'
    name: 'surf'
    line: 54
```

The `cause` field is empty in this case. Each field is described in more detail in the sections that follow.

Message Identifiers

A message identifier is a tag that you attach to an error or warning statement that makes that error or warning uniquely recognizable by MATLAB. You can use message identifiers with error reporting to better identify the source of an error, or with warnings to control any selected subset of the warnings in your programs.

The message identifier is a read-only character string that specifies a *component* and a *mnemonic* label for an error or warning. The format of a simple identifier is

```
component:mnemonic
```

A colon separates the two parts of the identifier: `component` and `mnemonic`. If the identifier uses more than one `mnemonic`, then additional colons are required to separate them. A message identifier must always contain at least one colon.

Some examples of message identifiers are

```
MATLAB:divideByZero
Simulink:actionNotTaken
TechCorp:OpenFile:notFoundInPath
```

Both the `component` and `mnemonic` fields must adhere to the following syntax rules:

- No white space (space or tab characters) is allowed anywhere in the identifier.
- The first character must be alphabetic, either uppercase or lowercase.
- The remaining characters can be alphanumeric or an underscore.

There is no length limitation to either the `component` or `mnemonic`. The identifier can also be an empty string.

Component Field. The `component` field specifies a broad category under which various errors and warnings can be generated. Common components are a particular product or toolbox name, such as `MATLAB` or `Control`, or perhaps the name of your company, such as `TechCorp` in the preceding example.

You can also use this field to specify a multilevel component. The following statement has a three-level component followed by a mnemonic label:

```
TechCorp:TestEquipDiv:Waveform:obsoleteSyntax
```

The component field enables you to guarantee the uniqueness of each identifier. Thus, while MATLAB uses the identifier `MATLAB:divideByZero` for its `'Divide by zero'` warning, you can reuse the `divideByZero` mnemonic by using your own unique component. For example,

```
warning('TechCorp:divideByZero', ...
       'A sprocket value was divided by zero.')
```

Mnemonic Field. The `mnemonic` field is a string normally used as a tag relating to the particular message. For example, when reporting an error resulting from the use of ambiguous syntax, a simple component and mnemonic such as the following might be appropriate:

```
MATLAB:ambiguousSyntax
```

Message Identifiers in an MException Object. When throwing an exception, create an appropriate identifier and save it to the `MException` object at the time you construct the object using the syntax

```
ME = MException(identifier, string)
```

For example,

```
ME = MException('AcctError:NoClient', ...
      'Client name not recognized.');

ME.identifier
ans =
    AcctError:NoClient
```

When responding to an exception, you can extract the message identifier from the `MException` object as shown here. Using the `surf` example again,

```
try
    surf
catch ME
    id = ME.identifier
end

id =
    MATLAB:nargchk:notEnoughInputs
```

Text of the Error Message

An error message in MATLAB is a read-only character string issued by the program code and returned in the `MException` object. This message can assist the user in determining the cause, and possibly the remedy, of the failure.

When throwing an exception, compose an appropriate error message and save it to the MException object at the time you construct the object using the syntax

```
ME = MException(identifier, string)
```

If your message string requires formatting specifications, like those available with the sprintf function, use this syntax for the MException constructor:

```
ME = MException(identifier, formatstring, arg1, arg2, ...)
```

For example,

```
S = 'Accounts';  f1 = 'ClientName';
ME = MException('AcctError:Incomplete', ...
      'Field ''%s.%s'' is not defined.', S, f1);

ME.message
ans =
    Field 'Accounts.ClientName' is not defined.
```

When responding to an exception, you can extract the error message from the MException object as follows:

```
try
    surf
catch ME
    msg = ME.message
end

msg =
    Not enough input arguments.
```

The Call Stack

The stack field of the MException object identifies the line number, function, and filename where the error was detected. If the error occurs in a called function, as in the following example, the stack field contains the line number, function name, and filename not only for the location of the immediate error, but also for each of the calling functions. In this case, stack

is an N-by-1 array, where N represents the depth of the call stack. That is, the stack field displays the M-file function name and line number where the exception occurred, the name and line number of the M-file caller, the caller's caller, etc., until the top-most M-file function is reached.

When throwing an exception, MATLAB stores call stack information in the `stack` field. You cannot write to this field; access is read-only.

For example, suppose you have three functions that reside in two separate M-files:

```
 mfileA.m
=========================
        .
        .
42 function A1(x, y)
43 B1(x, y);



 mfileB.m
=========================
        .
        .
 8 function B1(x, y)
 9 B2(x, y)
        .
        .
26 function B2(x, y)
27      .
28      .
29      .
30      .
31 %  Throw exception here
```

Catch the exception in variable `ME` and then examine the `stack` field:

```
for k=1:length(ME.stack)
    ME.stack(k)
end
```

```
ans =
    file: 'C:\matlab\test\mfileB.m'
    name: 'B2'
    line: 31
ans =
    file: 'C:\matlab\test\mfileB.m'
    name: 'B1'
    line: 9
ans =
    file: 'C:\matlab\test\mfileA.m'
    name: 'A1'
    line: 43
```

The Cause Array

In some situations, it can be important to record information about not only the one command that caused execution to stop, but also other exceptions that your code caught. You can save these additional `MException` objects in the `cause` field of the primary exception.

The `cause` field of an `MException` is an optional cell array of related `MException` objects. You must use the following syntax when adding objects to the `cause` cell array:

```
primaryException = addCause(primaryException, secondaryException)
```

This example attempts to assign an array `D` to variable `X`. If the `D` array does not exist, the code attempts to load it from a MAT-file and then retries assigning it to `X`. If the load fails, a new `MException` object (`ME3`) is constructed to store the cause of the first two errors (`ME1` and `ME2`):

```
try
    X = D(1:25)
catch ME1
    try
        filename = 'test200';
        load(filename);
        X = D(1:25)
    catch ME2
        ME3 = MException('MATLAB:LoadErr', ...
```

```
                'Unable to load from file %s', filename);
            ME3 = addCause(ME3, ME1);
            ME3 = addCause(ME3, ME2);
        end
    end
```

There are two exceptions in the cause field of ME3:

```
ME3.cause
ans =
    [1x1 MException]
    [1x1 MException]
```

Examine the cause field of ME3 to see the related errors:

```
ME3.cause{:}
ans =

 MException object with properties:

    identifier: 'MATLAB:UndefinedFunction'
       message: 'Undefined function or method 'D' for input
arguments of type 'double'.'
         stack: [0x1 struct]
         cause: {}
ans =

 MException object with properties:

    identifier: 'MATLAB:load:couldNotReadFile'
       message: 'Unable to read file test204: No such file or
directory.'
         stack: [0x1 struct]
         cause: {}
```

Methods of the MException Class

There are ten methods that you can use with the MException class. The names of these methods are case-sensitive. See the MATLAB function reference pages for more information.

Method Name	Description
addCause	Append an MException to the cause field of another MException.
disp	Display an MException object.
eq	Compare MException objects for equality.
getReport	Return a formatted message based on the current exception.
isequal	Compare MException objects for equality.
last	Return the last uncaught exception. This is a static method.
ne	Compare MException objects for inequality.
rethrow	Reissue an exception that has previously been caught.
throw	Issue an exception.
throwAsCaller	Issue an exception, but omit the current stack frame from the stack field.

Throwing an Exception

When your program detects a fault that will keep it from completing as expected or will generate erroneous results, you should halt further execution and report the error by throwing an exception. The basic steps to take are

- Detect the error. This is often done with some type of conditional statement, such as an `if` statement that checks the output of the current operation.
- Construct an `MException` object to represent the error. Add a message identifier string and error message string to the object when calling the constructor.
- If there are other exceptions that may have contributed to the current error, you can store the `MException` object for each in the `cause` field of a single `MException` that you intend to throw. Use the `addCause` method for this.
- Use the `throw` or `throwAsCaller` function to have the MATLAB software issue the exception. At this point, MATLAB stores call stack information in the `stack` field of the `MException`, exits the currently running function, and returns control to either the keyboard or an enclosing catch block in a calling function.

Responding to an Exception

In this section…

Overview

As stated earlier, the MATLAB software, by default, terminates the currently running program when an exception is thrown. If you catch the exception in your program, however, you can capture information about what went wrong, and deal with the situation in a way that is appropriate for the particular condition. This requires a `try-catch` statement.

This section covers the following topics:

The try-catch Statement

When you have statements in your code that could generate undesirable results, put those statements into a `try-catch` block that catches any errors and handles them appropriately.

A `try-catch` statement looks something like the following pseudocode. It consists of two parts:

- A `try` block that includes all lines between the `try` and `catch` statements.
- A `catch` block that includes all lines of code between the `catch` and `end` statements.

```
   try
      Perform one ...
         or more operations
A  catch ME
      Examine error info in exception object ME
      Attempt to figure out what went wrong
      Either attempt to recover, or clean up and abort
```

```
      end

  B   Program continues
```

The program executes the statements in the `try` block. If it encounters an error, it skips any remaining statements in the `try` block and jumps to the start of the `catch` block (shown here as point `A`). If all operations in the `try` block succeed, then execution skips the `catch` block entirely and goes to the first line following the `end` statement (point `B`).

Specifying the `try`, `catch`, and `end` commands and also the code of the `try` and `catch` blocks on separate lines is recommended. If you combine any of these components on the same line, separate them with commas:

```
try, surf, catch ME, ME.stack, end
ans =
    file: 'matlabroot\toolbox\matlab\graph3d\surf.m'
    name: 'surf'
    line: 54
```

You cannot define nested functions within a `try` or `catch` block.

The Try Block

On execution, your code enters the `try` block and executes each statement as if it were part of the regular program. If no errors are encountered, MATLAB skips the `catch` block entirely and continues execution following the `end` statement. If any of the `try` statements fail, MATLAB immediately exits the `try` block, leaving any remaining statements in that block unexecuted, and enters the `catch` block.

The Catch Block

The `catch` command marks the start of a `catch` block and provides access to a data structure that contains information about what caused the exception. This is shown as the variable `ME` in the preceding pseudocode. This data structure is an object of the MATLAB `MException` class. When an exception occurs, MATLAB constructs an instance of this class and returns it in the `catch` statement that handles that error.

You are not required to specify any argument with the `catch` statement. If you do not need any of the information or methods provided by the `MException` object, just specify the `catch` keyword alone.

The `MException` object is constructed by internal code in the program that fails. The object has properties that contain information about the error that can be useful in determining what happened and how to proceed. The `MException` object also provides access to methods that enable you to respond to the exception. See the section on"The MException Class" on page 8-5 to find out more about the `MException` class.

Having entered the `catch` block, MATLAB executes the statements in sequence. These statements can attempt to

- Attempt to resolve the error.
- Capture more information about the error.
- Switch on information found in the `MException` object and respond appropriately.
- Clean up the environment that was left by the failing code.

The `catch` block often ends with a `rethrow` command. The `rethrow` causes MATLAB to exit the current function, keeping the call stack information as it was when the exception was first thrown. If this function is at the highest level, that is, it was not called by another function, the program terminates. If the failing function was called by another function, it returns to that function. Program execution continues to return to higher level functions, unless any of these calls were made within a higher-level `try` block, in which case the program executes the respective catch block.

More information about the `MException` class is provided in the section "Capturing Information About the Error" on page 8-5.

Suggestions on How to Handle an Exception

The following example reads the contents of an image file. The `try` block attempts to open and read the file. If either the open or read fails, the program catches the resulting exception and saves the `MException` object in the variable `ME1`.

The catch block in the example checks to see if the specified file could not be found. If so, the program allows for the possibility that a common variation of the filename extension (e.g., jpeg instead of jpg) was used by retrying the operation with a modified extension. This is done using a try-catch statement nested within the original try-catch.

```
function d_in = read_image(filename)
[path name ext] = fileparts(filename);
try
   fid = fopen(filename, 'r');
   d_in = fread(fid);
catch ME1
   % Get last segment of the error message identifier.
   idSegLast = regexp(ME1.identifier, '(?<=:)\w+$', 'match');

   % Did the read fail because the file could not be found?
   if strcmp(idSegLast, 'InvalidFid') && ...
      ~exist(filename, 'file')

      % Yes. Try modifying the filename extension.
      switch ext
      case '.jpg'    % Change jpg to jpeg
          filename = strrep(filename, '.jpg', '.jpeg')
      case '.jpeg'   % Change jpeg to jpg
          filename = strrep(filename, '.jpeg', '.jpg')
      case '.tif'    % Change tif to tiff
          filename = strrep(filename, '.tif', '.tiff')
      case '.tiff'   % Change tiff to tif
          filename = strrep(filename, '.tiff', '.tif')
      otherwise
         fprintf('File %s not found\n', filename);
         rethrow(ME1);
      end

      % Try again, with modifed filenames.
      try
         fid = fopen(filename, 'r');
         d_in = fread(fid);
      catch ME2
         fprintf('Unable to access file %s\n', filename);
```

```
            ME2 = addCause(ME2, ME1);
            rethrow(ME2)
        end
    end
end
```

This example illustrates some of the actions that you can take in response to an exception:

- Compare the `identifier` field of the `MException` object against possible causes of the error.
- Use a nested `try-catch` statement to retry the open and read operations using a known variation of the filename extension.
- Display an appropriate message in the case that the file truly does not exist and then rethrow the exception.
- Add the first `MException` object to the `cause` field of the second.
- Rethrow the exception. This stops program execution and displays the error message.

Cleaning up any unwanted results of the error is also advisable. For example, your program may have allocated a significant amount of memory that it no longer needs.

Warnings

In this section...

Reporting a Warning

Like `error`, the `warning` function alerts the user of unexpected conditions detected when running a program. However, `warning` does not halt the execution of the program. It displays the specified warning message and then continues.

Use `warning` in your code to generate a warning message during execution. Specify the message string as the input argument to `warning`. For example,

```
warning('Input must be a string')
```

Warnings also differ from errors in that you can disable any warnings that you do not want to see. You do this by invoking `warning` with certain control parameters. See "Warning Control" on page 8-24 for more information.

Formatted Message Strings

The warning message string you specify can contain formatting conversion characters, such as those used with the MATLAB `sprintf` function. Make the warning string the first argument, and add any variables used by the conversion as subsequent arguments.

```
warning('formatted_warningmsg', arg1, arg2, ...)
```

For example, if your program cannot process a given parameter, you might report a warning with

```
warning('Ambiguous parameter name, "%s".', param)
```

MATLAB converts special characters like `%d` and `%s` in the warning message string only when you specify more than one input argument with `warning`. See "Formatted Message Strings" on page 8-22 for information.

Message Identifiers

Use a message identifier argument with `warning` to attach a unique tag to a warning message. MATLAB uses this tag to better identify the source of a warning. The first argument in this example is the message identifier.

```
warning('MATLAB:paramAmbiguous', ...
        'Ambiguous parameter name, "%s".', param)
```

See "Warning Control Statements" on page 8-26 for more information on how to use identifiers with warnings.

Identifying the Cause

The `lastwarn` function returns a string containing the last warning message issued by MATLAB. Use this to enable your program to identify the cause of a warning that has just been issued. To return the most recent warning message to the variable `warnmsg`, type

```
warnmsg = lastwarn;
```

You can also change the text of the last warning message with a new message or with an empty string as shown here:

```
lastwarn('newwarnmsg');   % Replace last warning with new string
lastwarn('');            % Replace last warning with empty string
```

Warning Control

In this section...

Overview

The MATLAB software gives you the ability to control what happens when a warning is encountered during M-file program execution. Options that are available include

- Display selected warnings.
- Ignore selected warnings.
- Stop in the debugger when a warning is invoked.
- Display an M-stack trace after a warning is invoked.

Depending on how you set your warning controls, you can have these actions affect all warnings in your code, specific warnings that you select, or just the most recently invoked warning.

Setting up this system of warning control involves several steps.

1. Start by determining the scope of the control you need for the warnings generated by your code. Do you want the control operations to affect all the warnings in your code at once, or do you want to be able to control certain warnings separately?

2. If the latter is true, you will need to identify those warnings you want to selectively control. This requires going through your code and attaching unique *message identifiers* to each of those warnings. If, on the other

hand, you do not require that fine a granularity of control, the warning statements in your code need no message identifiers.

3 When you are ready to run your programs, use the MATLAB warning control statements to exercise the desired controls on all or selected warnings. Include message identifiers in these control statements when selecting specific warnings to act upon.

Warning Statements

The warning statements you put into your M-file code must contain the string to be displayed when the warning is incurred, and may also contain a message identifier. If you are not planning to use warning control or if you do not need to single out certain warnings for control, you need to specify only the message string. Use the syntax shown in "Warnings" on page 8-22. Valid formats are

```
warning('warnmsg')
warning('formatted_warnmsg', arg1, arg2, ...)
```

Attaching an Identifier to the Warning Statement

If you want to be able to apply control statements to specific warnings, you need to include a message identifier in the warning statements you wish to control. The message identifier must be the first argument in the statement. Valid formats are

```
warning('msg_id', 'warnmsg')
warning('msg_id', 'formatted_warnmsg', arg1, arg2, ...)
```

See "Message Identifiers" on page 8-8 for information on how to specify the `msg_id` argument.

Note When you specify more than one input argument with `warning`, MATLAB treats the `warnmsg` string as if it were a `formatted_warnmsg`. This is explained in "Formatted Message Strings" on page 8-22.

Warning Control Statements

Once you have the warning statements in your M-file and are ready to execute it, you tell MATLAB how to act on these warnings by issuing control statements. These statements place the specified warning(s) into a desired state and have the format

```
warning state msg_id
```

Control statements can return information on the state of selected warnings if you assign the output to a variable, as shown below. See "Output from Control Statements" on page 8-28.

```
s = warning('state', 'msg_id');
```

Warning States

There are three possible values for the `state` argument of a warning control statement.

State	Description
`on`	Enable the display of selected warning message.
`off`	Disable the display of selected warning message.
`query`	Display the current state of selected warning.

Message Identifiers

In addition to the message identifiers already discussed, there are three other identifiers that you can use in control statements only.

Identifier	Description
`msg_id string`	Set selected warning to the specified state.
`all`	Set all warnings to the specified state.
`last`	Set only the last displayed warning to the specified state.

Note MATLAB starts up with all warnings enabled, except for those displayed in response to the command, `warning('query', 'all')`.

Example 1 — Enabling a Selected Warning

Enable just the `actionNotTaken` warning from Simulink by first turning off all warnings and then setting just that warning to `on`.

```
warning off all
warning on Simulink:actionNotTaken
```

Next, use `query` to determine the current state of all warnings. It reports that you have set all warnings to `off`, with the exception of `Simulink:actionNotTaken`.

```
warning query all
The default warning state is 'off'. Warnings not set to the
default are

  State  Warning Identifier

     on  Simulink:actionNotTaken
```

Example 2 — Disabling the Most Recent Warning

Evaluating `inv` on zero displays a warning message. Turn off the most recently invoked warning with `warning off last`.

```
inv(0)
Warning: Matrix is singular to working precision.
ans =
   Inf

warning off last

inv(0)              % No warning is displayed this time
ans =
   Inf
```

Output from Control Statements

The `warning` function, when used in a control statement, returns a MATLAB structure array containing the previous state of the selected warning(s). Use the following syntax to return this information in structure array `s`:

```
s = warning('state', 'msg_id');
```

You must type the command using the MATLAB function format; parentheses and quotation marks are required.

Note MATLAB does not display `warning` output if you do not assign the output to a variable.

The next example turns off `divideByZero` warnings for the MATLAB component, and returns the `identifier` and previous `state` in a 1-by-1 structure array.

```
s = warning('off','MATLAB:divideByZero')
s =
    identifier: 'MATLAB:divideByZero'
         state: 'on'
```

You can use output variables with any type of warning control statement. If you just want to collect the information but do not want to change state, simply perform a `query` on the warning(s). MATLAB returns the current state of those warnings selected by the message identifier.

```
s = warning('query', 'msg_id');
```

If you want to change state, but save the former state so you can restore it later, use the return structure array to save that state. The following example does an implicit `query`, returning state information in `s`, and then turns on all warnings.

```
s = warning('on', 'all');
```

See "Saving and Restoring State" on page 8-30, for more information on restoring the former state of warnings.

Output Structure Array

Each element of the structure array returned by `warning` contains two fields.

Field Name	Description
`identifier`	Message identifier string, `'all'`, or `'last'`
`state`	State of warning(s) prior to invoking this control statement

If you query for the state of just one warning, using a message identifier or `'last'` in the command, MATLAB returns a one-element structure array. The `identifier` field contains the selected message identifier, and the `state` field holds the current state of that warning:

```
s = warning('query','last')
s =
   identifier: 'MATLAB:divideByZero'
        state: 'on'
```

If you query for the state of all warnings, using `'all'` in the command, MATLAB returns a structure array having one or more elements:

- The first element of the array always represents the default state. (This is the state set by the last `warning on|off all` command.)
- Each other element of the array represents a warning that is in a state different from the default.

```
warning off all
warning on MATLAB:divideByZero
warning on MATLAB:fileNotFound

s = warning('query', 'all')
s =
   3x1 struct array with fields:
       identifier
       state

s(1)
ans =
```

```
      identifier: 'all'
           state: 'off'

   s(2)
   ans =
      identifier: 'MATLAB:divideByZero'
           state: 'on'

   s(3)
   ans =
      identifier: 'MATLAB:fileNotFound'
           state: 'on'
```

Saving and Restoring State

To temporarily change the state of some warnings and then later return to your original settings, save the original state in a structure array and then restore it from that array. You can save and restore the state of all of your warnings or just one that you select with a message identifier.

To save the current warning state, assign the output of a warning control statement, as discussed in "Output from Control Statements" on page 8-28. The following statement saves the current state of all warnings in structure array s:

```
s = warning('query', 'all');
```

To restore state from s, use the syntax shown below. Note that the MATLAB function format (enclosing arguments in parentheses) is required.

```
warning(s)
```

Example 1 — Performing an Explicit Query

Perform a query of all warnings to save the current state in structure array s:

```
s = warning('query', 'all');
```

Then, after doing some work that includes making changes to the state of some warnings, restore the original state of all warnings:

```
warning(s)
```

Example 2 — Performing an Implicit Query

Turn on one particular warning, saving the previous state of this warning in `s`. Remember that this nonquery syntax (where `state` equals `on` or `off`) performs an implicit query prior to setting the new state:

```
s = warning('on', 'Control:parameterNotSymmetric');
```

Restore the state of that one warning when you are ready, with

```
warning(s)
```

Backtrace and Verbose Modes

In addition to warning messages, there are two *modes* that can be enabled or disabled with a warning control statement. These modes are shown here.

Mode	Description	Default
`backtrace`	Display an M-stack trace after a warning is invoked.	`on (enabled)`
`verbose`	Display a message on how to suppress the warning.	`off (terse)`

The syntax for this type of control statement is as follows, where `state`, in this case, can be only `on`, `off`, or `query`:

```
warning state mode
```

Note that there is no need to include a message identifier with this type of control statement. All enabled warnings are affected by the this type of control statement.

Note You cannot save and restore the current state of the `backtrace` or `verbose` modes as you can with other states.

Example 1 — Displaying a Stack Trace on a Specific Warning

It can be difficult to locate the source of a warning when it is generated from code buried in several levels of function calls. This example generates

a warning within a function that is nested several levels deep within the primary function in file `f1.m`:

```
function f1(a, b)
   for k = a:-1:b
       f2(k)
   end
   function f2(x)
       f3(x-1)
       function f3(y)
           x = log(y);
       end
   end
end
```

After enabling all warnings, run the M-file. The code generates a `Log of zero` warning. In an M-file of this size, it is not difficult to find the cause of the warning, but in an M-file of several hundred lines, this could take some time:

```
warning on all

f1(50,1)
Warning: Log of zero.
```

To simplify the debug process, enable backtrace mode. In this mode, MATLAB reports which function generated the warning (`f3`), the line number of the attempted operation (line `8`), the sequence of function calls that led up to the execution of the function (`f1>f2/f3`), and the line at which each of these function call was made (`3` and `6`):

```
warning on backtrace
f1(50,1)
Warning: Log of zero.
> In f1>f2/f3 at 8
  In f1>f2 at 6
  In f1 at 3
```

Example 2 — Enabling Verbose Warnings

When you enable verbose warnings, MATLAB displays an extra line of information with each warning that tells you how to suppress it:

Turn on all warnings, disable backtrace (if you have just run the previous example), and enable verbose warnings:

```
warning on all
warning off backtrace
warning on verbose
```

Call the function described in Example 1 to find out how to suppress any warnings generated by that function:

```
f1(50,1)
Warning: Log of zero.
(Type "warning off MATLAB:log:logOfZero" to suppress this warning.)
```

Use the message identifier `MATLAB:log:logOfZero` to disable only this warning, and run the function again. This time the warning message is not displayed:

```
warning off MATLAB:log:logOfZero

f1(50,1)
```

Debugging Errors and Warnings

You can direct the MATLAB software to temporarily stop the execution of an M-file in the event of a run-time error or warning, at the same time opening a debug window paused at the M-file line that generated the error or warning. This enables you to examine values internal to the program and determine the cause of the error.

Use the `dbstop` function to have MATLAB stop execution and enter debug mode when any M-file you subsequently run produces a run-time error or warning. There are three types of such breakpoints that you can set.

Command	Description
`dbstop if all error`	Stop on any error.
`dbstop if error`	Stop on any error not detected within a `try-catch` block.
`dbstop if warning`	Stop on any warning.

In all three cases, the M-file you are trying to debug must be in a directory that is on the search path or in the current directory.

You cannot resume execution after an error; use `dbquit` to exit from the Debugger. To resume execution after a warning, use `dbcont` or `dbstep`.

Program Scheduling

Using a MATLAB Timer Object

In this section...

Overview

The MATLAB software includes a timer object that you can use to schedule the execution of MATLAB commands. This section describes how you can create timer objects, start a timer running, and specify the processing that you want performed when a timer fires. A timer is said to *fire* when the amount of time specified by the timer object elapses and the timer object executes the commands you specify.

To use a timer, perform these steps:

1. Create a timer object.

   You use the `timer` function to create a timer object. See "Creating Timer Objects" on page 9-5 for more information.

2. Specify which MATLAB commands you want executed when the timer fires and control other aspects of timer object behavior.

   You use timer object properties to specify this information. To learn about all the properties supported by the timer object, see "Working with Timer Object Properties" on page 9-7. (You can also set timer object properties when you create them, in step 1.)

3. Start the timer object.

   After you create the timer object, you must start it, using either the `start` or `startat` function. See "Starting and Stopping Timers" on page 9-10 for more information.

4. Delete the timer object when you are done with it.

After you are finished using a timer object, you should delete it from memory. See "Deleting Timer Objects from Memory" on page 9-23 for more information.

Note The specified execution time and the actual execution of a timer can vary because timer objects work in the MATLAB single-threaded execution environment. The length of this time lag is dependent on what other processing MATLAB is performing. To force the execution of the callback functions in the event queue, include a call to the `drawnow` function in your code. The `drawnow` function flushes the event queue.

Example: Displaying a Message

The following example sets up a timer object that executes a MATLAB command string after 10 seconds elapse. The example creates a timer object, specifying the values of two timer object properties, `TimerFcn` and `StartDelay`. `TimerFcn` specifies the timer callback function. This is the MATLAB command string or M-file that you want to execute when the timer fires. In the example, the timer callback function sets the value of the MATLAB workspace variable `stat` and executes the MATLAB `disp` command. The `StartDelay` property specifies how much time elapses before the timer fires.

After creating the timer object, the example uses the `start` function to start the timer object. (The additional commands in this example are included to illustrate the timer but are not required for timer operation.)

```
t = timer('TimerFcn', 'stat=false; disp(''Timer!'')',...
                 'StartDelay',10);
start(t)

stat=true;
while(stat==true)
  disp('.')
  pause(1)
end
```

When you execute this code, it produces this output:

```
.
.
.
.
.
.
.
.
.
Timer!

delete(t) % Always delete timer objects after using them.
```

Creating Timer Objects

In this section...

Creating the Object

To use a timer in MATLAB, you must create a timer object. The timer object represents the timer in MATLAB, supporting various properties and functions that control its behavior.

To create a timer object, use the `timer` function. This creates a valid timer object with default values for most properties. The following shows an example of the default timer object and its summary display:

```
t = timer
Timer Object: timer-1

   Timer Settings
      ExecutionMode: singleShot
             Period: 1
           BusyMode: drop
            Running: off

   Callbacks
           TimerFcn: ''
           ErrorFcn: ''
           StartFcn: ''
            StopFcn: ''
```

MATLAB names the timer object `timer-1`. (See "Naming the Object" on page 9-6 for more information.)

To specify the value of timer object properties after you create it, you can use the `set` function. This example sets the value of the `TimerFcn` property and the `StartDelay` property. For more information about timer object properties, see "Working with Timer Object Properties" on page 9-7.

```
set(t,'TimerFcn','disp(''Hello World!'')','StartDelay',5)
```

You can also set timer object properties when you create the timer object by specifying property name and value pairs as arguments to the `timer` function. The following example sets the same properties at object creation time:

```
t = timer('TimerFcn', 'disp(''Hello World!'')','StartDelay',5);
```

Always delete timer objects when you are done using them. See "Deleting Timer Objects from Memory" on page 9-23 for more information.

Naming the Object

MATLAB assigns a name to each timer object you create. This name has the form `'timer-`*`i`*`'`, where *`i`* is a number representing the total number of timer objects created this session.

For example, the first time you call the timer function to create a timer object, MATLAB names the object `timer-1`. If you call the timer function again to create another timer object, MATLAB names the object `timer-2`.

MATLAB keeps incrementing the number associated with each timer object it creates, even if you delete the timer objects you already created. For example, if you delete the first two timer objects and create a new object, MATLAB names it `timer-3`, even though the other two timer objects no longer exist in memory. To reset the numeric part of timer object names to 1, execute the `clear classes` command.

Working with Timer Object Properties

In this section...

To get information about timer object properties, see the `timer` function reference page.

Retrieving the Value of Timer Object Properties

The timer object supports many properties that provide information about the current state of the timer object and control aspects of its functioning. To retrieve the value of a timer object property, you can use the `get` function or use subscripts (dot notation) to access the field.

The following example uses the `set` function to retrieve the value of the `ExecutionMode` property:

```
t = timer;

tmode = get(t,'ExecutionMode')

tmode =

singleShot
```

The following example uses dot notation to retrieve the value of the `ExecutionMode` property:

```
tmode = t.ExecutionMode

tmode =

singleShot
```

To view a list of all the properties of a timer object, use the `get` function, specifying the timer object as the only argument:

```
get(t)
       AveragePeriod: NaN
            BusyMode: 'drop'
            ErrorFcn: ''
       ExecutionMode: 'singleShot'
       InstantPeriod: NaN
                Name: 'timer-4'
    ObjectVisibility: 'on'
              Period: 1
             Running: 'off'
          StartDelay: 0
            StartFcn: ''
             StopFcn: ''
                 Tag: ''
       TasksExecuted: 0
      TasksToExecute: Inf
            TimerFcn: ''
                Type: 'timer'
            UserData: []
```

Setting the Value of Timer Object Properties

To set the value of a timer object property, use the `set` function or subscripted assignment (dot notation). You can also set timer object properties when you create the timer object. For more information, see “Creating Timer Objects” on page 9-5.

The following example uses both methods to assign values to timer object properties. The example creates a timer that, once started, displays a message every second until you stop it with the `stop` command.

1 Create a timer object.

```
t = timer;
```

2 Assign values to timer object properties using the `set` function.

```
set(t,'ExecutionMode','fixedRate','BusyMode','drop','Period',1);
```

3 Assign a value to the timer object `TimerFcn` property using dot notation.

```
t.TimerFcn = 'disp(''Processing...'')'
```

4 Start the timer object. It displays a message at 1-second intervals.

```
start(t)
```

5 Stop the timer object.

```
stop(t)
```

6 Delete timer objects after you are done using them.

```
delete(t)
```

Viewing a List of All Settable Properties

To view a list of all timer object properties that can have values assigned to them (in contrast to the read-only properties), use the `set` function, specifying the timer object as the only argument.

The display includes the values you can use to set the property if, like the `BusyMode` property, the property accepts an enumerated list of values.

```
t = timer;

set(t)
   BusyMode: [ {drop} | queue | error ]
   ErrorFcn: string -or- function handle -or- cell array
   ExecutionMode: [{singleShot} | fixedSpacing | fixedDelay | fixedRate]
   Name
   ObjectVisibility: [ {on} | off ]
   Period
   StartDelay
   StartFcn: string -or- function handle -or- cell array
   StopFcn: string -or- function handle -or- cell array
   Tag
   TasksToExecute
   TimerFcn: string -or- function handle -or- cell array
   UserData
```

Starting and Stopping Timers

In this section...

Note Because the timer works within the MATLAB single-threaded environment, it cannot guarantee execution times or execution rates.

Starting a Timer

To start a timer object, call the `start` function, specifying the timer object as the only argument. The `start` function starts a timer object running; the amount of time the timer runs is specified in seconds in the `StartDelay` property.

This example creates a timer object that displays a greeting after 5 seconds elapse.

1 Create a timer object, specifying values for timer object properties.

```
t = timer('TimerFcn','disp(''Hello World!'')','StartDelay', 5);
```

2 Start the timer object.

```
start(t)
```

3 Delete the timer object after you are finished using it.

```
delete(t);
```

Starting a Timer at a Specified Time

To start a timer object and specify a date and time for the timer to fire, (rather than specifying the number of seconds to elapse), use the `startat` function. This function starts a timer object and allows you to specify the date, hour,

minute, and second when you want to the timer to execute. You specify the time as a MATLAB serial date number or as a specially formatted date text string.

This example creates a timer object that displays a message after an hour has elapsed. The `startat` function starts the timer object running and calculates the value of the `StartDelay` property based on the time you specify.

```
t2=timer('TimerFcn','disp(''It has been an hour now.'')');

startat(t2,now+1/24);
```

Stopping Timer Objects

Once started, the timer object stops running if one of the following conditions apply:

- The timer function callback (`TimerFcn`) has been executed the number of times specified in the `TasksToExecute` property.
- An error occurred while executing a timer function callback (`TimerFcn`).

You can also stop a timer object by using the `stop` function, specifying the timer object as the only argument. The following example illustrates stopping a timer object:

1. Create a timer object.

   ```
   t = timer('TimerFcn','disp(''Hello World!'')', ...
             'StartDelay', 100);
   ```

2. Start it running.

   ```
   start(t)
   ```

3. Check the state of the timer object after starting it.

   ```
   get(t,'Running')

   ans =

   on
   ```

4 Stop the timer using the `stop` command and check the state again. When a timer stops, the value of the `Running` property of the timer object is set to `'off'`.

```
stop(t)

get(t,'Running')

ans =

off
```

5 Delete the timer object when you are finished using it.

```
delete(t)
```

Note The timer object can execute a callback function that you specify when it starts or stops. See "Creating and Executing Callback Functions" on page 9-14.

Blocking the MATLAB Command Line

By default, when you use the `start` or `startat` function to start a timer object, the function returns control to the command line immediately. For some applications, you might prefer to block the command line until the timer fires. To do this, call the `wait` function right after calling the `start` or `startat` function.

1 Create a timer object.

```
t = timer('StartDelay', 5,'TimerFcn', ...
          'disp(''Hello World!'')');
```

2 Start the timer object running.

```
start(t)
```

3 After the `start` function returns, call the `wait` function immediately. The `wait` function blocks the command line until the timer object fires.

```
wait(t)
```

4 Delete the timer object after you are finished using it.

```
delete(t)
```

Creating and Executing Callback Functions

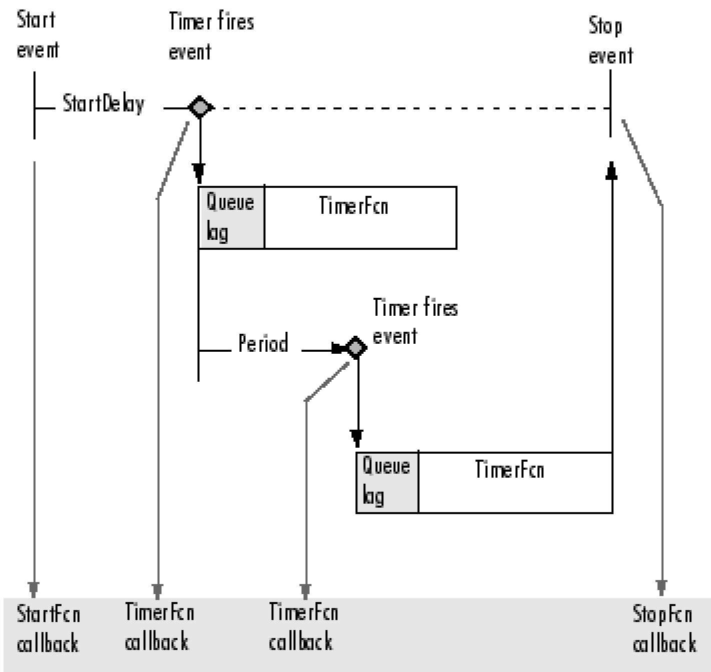
In this section...

Note Callback function execution might be delayed if the callback involves a CPU-intensive task such as updating a figure.

Associating Commands with Timer Object Events

The timer object supports properties that let you specify the MATLAB commands that execute when a timer fires, and for other timer object events, such as starting, stopping, or when an error occurs. These are called *callbacks*. To associate MATLAB commands with a timer object event, set the value of the associated timer object callback property.

The following diagram shows when the events occur during execution of a timer object and give the names of the timer object properties associated with each event. For example, to associate MATLAB commands with a start event, assign a value to the `StartFcn` callback property. Error callbacks can occur at any time.



Timer Object Events and Related Callback Function

Creating Callback Functions

When the time period specified by a timer object elapses, the timer object executes one or more MATLAB functions of your choosing. You can specify the functions directly as the value of the callback property. You can also put the commands in an M-file function and specify the M-file function as the value of the callback property.

Specifying Callback Functions Directly

This example creates a timer object that displays a greeting after 5 seconds. The example specifies the value of the `TimerFcn` callback property directly, putting the commands in a text string.

```
t = timer('TimerFcn','disp(''Hello World!'')','StartDelay',5);
```

Note When you specify the callback commands directly as the value of the callback function property, the commands are evaluated in the MATLAB workspace.

Putting Commands in a Callback Function

Instead of specifying MATLAB commands directly as the value of a callback property, you can put the commands in an M-file and specify the M-file as the value of the callback property.

When you create a callback function, the first two arguments must be a handle to the timer object and an event structure. An event structure contains two fields: `Type` and `Data`. The `Type` field contains a text string that identifies the type of event that caused the callback. The value of this field can be any of the following strings: `'StartFcn'`, `'StopFcn'`, `'TimerFcn'`, or `'ErrorFcn'`. The `Data` field contains the time the event occurred.

In addition to these two required input arguments, your callback function can accept application-specific arguments. To receive these input arguments, you must use a cell array when specifying the name of the function as the value of a callback property. For more information, see "Specifying the Value of Callback Function Properties" on page 9-17.

Example: Writing a Callback Function

This example implements a simple callback function that displays the type of event that triggered the callback and the time the callback occurred. To illustrate passing application-specific arguments, the example callback function accepts as an additional argument a text string and includes this text string in the display output. To see this function used with a callback property, see "Specifying the Value of Callback Function Properties" on page 9-17.

```
function my_callback_fcn(obj, event, string_arg)

txt1 = ' event occurred at ';
txt2 = string_arg;

event_type = event.Type;
```

```
event_time = datestr(event.Data.time);

msg = [event_type txt1 event_time];
disp(msg)
disp(txt2)
```

Specifying the Value of Callback Function Properties

You associate a callback function with a specific event by setting the value of the appropriate callback property. You can specify the callback function as a text string, cell array, or function handle. To access the object and event arguments, you must specify the function as a cell array or as a function handle. If your callback function accepts additional arguments, you must use a cell array.

The following table shows the syntax for several sample callback functions and describes how you call them.

Callback Function Syntax	How to Specify as a Property Value
`function myfile`	`set(h, 'StartFcn', 'myfile')`
`function myfile(obj, event)`	`set(h, 'StartFcn', @myfile)`
`function myfile(obj, event, arg1, arg2)`	`set(h, 'StartFcn', {'myfile', 5, 6})`
`function myfile(obj, event, arg1, arg2)`	`set(h, 'StartFcn', {@myfile, 5, 6})`

This example illustrates several ways you can specify the value of timer object callback function properties, some with arguments and some without. To see the code of the callback function, `my_callback_fcn`, see "Example: Writing a Callback Function" on page 9-16.

1 Create a timer object.

```
t = timer('StartDelay', 4,'Period', 4,'TasksToExecute', 2,...
          'ExecutionMode','fixedRate');
```

2 Specify the value of the StartFcn callback. Note that the example specifies the value in a cell array because the callback function needs to access arguments passed to it.

```
t.StartFcn = {'my_callback_fcn', 'My start message'};
```

3 Specify the value of the StopFcn callback. The example specifies the callback function by its handle, rather than as a text string. Again, the value is specified in a cell array because the callback function needs to access the arguments passed to it.

```
t.StopFcn = { @my_callback_fcn, 'My stop message'};
```

4 Specify the value of the TimerFcn callback. The example specifies the MATLAB commands in a text string.

```
t.TimerFcn = 'disp(''Hello World!'')';
```

5 Start the timer object.

```
start(t)
```

The example outputs the following.

```
StartFcn event occurred at 10-Mar-2004 17:16:59
Start message
Hello World!
Hello World!
StopFcn event occurred at 10-Mar-2004 17:16:59
Stop message
```

6 Delete the timer object after you are finished with it.

```
delete(t)
```

Timer Object Execution Modes

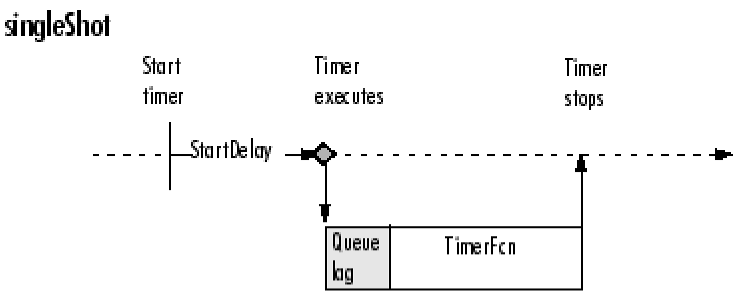
In this section...

Executing a Timer Callback Function Once

The timer object supports several execution modes that determine how it schedules the timer callback function (`TimerFcn`) for execution. You specify the execution mode by setting the value of the `ExecutionMode` property.

To execute a timer callback function once, set the `ExecutionMode` property to `'singleShot'`. This is the default execution mode. In this mode, the timer object starts the timer and, after the time period specified in the `StartDelay` property elapses, adds the timer callback function (`TimerFcn`) to the MATLAB execution queue. When the timer callback function finishes, the timer stops.

The following figure graphically illustrates the parts of timer callback execution for a `singleShot` execution mode. The shaded area in the figure, labelled queue lag, represents the indeterminate amount of time between when the timer adds a timer callback function to the MATLAB execution queue and when the function starts executing. The duration of this lag is dependent on what other processing MATLAB happens to be doing at the time.



Timer Callback Execution (singleShot Execution Mode)

Executing a Timer Callback Function Multiple Times

The timer object supports three multiple-execution modes:

- `'fixedRate'`
- `'fixedDelay'`
- `'fixedSpacing'`

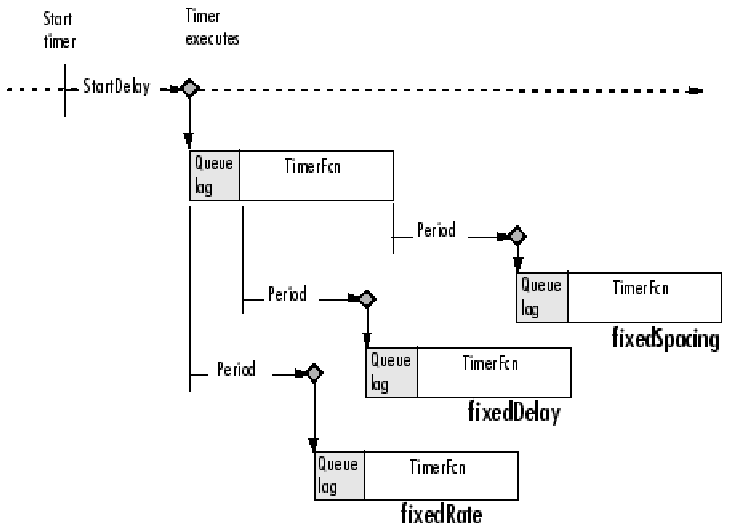
In many ways, these execution modes operate the same:

- The `TasksToExecute` property specifies the number of times you want the timer to execute the timer callback function (`TimerFcn`).
- The `Period` property specifies the amount of time between executions of the timer callback function.
- The `BusyMode` property specifies how the timer object handles queuing of the timer callback function when the previous execution of the callback function has not completed. See "Handling Callback Function Queuing Conflicts" on page 9-21 for more information.

The execution modes differ only in where they start measuring the time period between executions. The following table describes these differences.

Execution Mode	Description
`'fixedRate'`	Time period between executions begins immediately after the timer callback function is added to the MATLAB execution queue.
`'fixedDelay'`	Time period between executions begins when the timer function callback actually starts executing, after any time lag due to delays in the MATLAB execution queue.
`'fixedSpacing'`	Time period between executions begins when the timer callback function finishes executing.

The following figure illustrates the difference between these modes. Note that the amount of time between executions (specified by the `Period` property) remains the same. Only the point at which execution begins is different.



Differences Between Execution Modes

Handling Callback Function Queuing Conflicts

At busy times, in multiple-execution scenarios, the timer may need to add the timer callback function (`TimerFcn`) to the MATLAB execution queue before the previously queued execution of the callback function has completed. You can determine how the timer object handles this scenario by using the `BusyMode` property.

If you specify `'drop'` as the value of the `BusyMode` property, the timer object skips the execution of the timer function callback if the previously scheduled callback function has not already completed.

If you specify `'queue'`, the timer object waits until the currently executing callback function finishes before queuing the next execution of the timer callback function.

Note In `'queue'` mode, the timer object tries to make the average time between executions equal the amount of time specified in the `Period` property. If the timer object has to wait longer than the time specified in the `Period` property between executions of the timer function callback, it shortens the time period for subsequent executions to make up the time.

If the `BusyMode` property is set to `'error'`, the timer object stops and executes the timer object error callback function (`ErrorFcn`), if one is specified.

Deleting Timer Objects from Memory

In this section...

Deleting One or More Timer Objects

When you are finished with a timer object, delete it from memory using the `delete` function:

```
delete(t)
```

When you delete a timer object, workspace variables that referenced the object remain. Deleted timer objects are invalid and cannot be reused. Use the `clear` command to remove workspace variables that reference deleted timer objects.

To remove all timer objects from memory, enter

```
delete(timerfind)
```

For information about the `timerfind` function, see "Finding Timer Objects in Memory" on page 9-24.

Testing the Validity of a Timer Object

To test if a timer object has been deleted, use the `isvalid` function. The `isvalid` function returns logical `0` (`false`) for deleted timer objects:

```
isvalid(t)
ans =

     0
```

Finding Timer Objects in Memory

In this section...

Finding All Timer Objects

To find all the timer objects that exist in memory, use the `timerfind` function. This function returns an array of timer objects. If you leave off the semicolon, and there are multiple timer objects in the array, `timerfind` displays summary information in a table:

```
t1 = timer;
t2 = timer;
t3 = timer;
t_array = timerfind

Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:    Name:
   1       singleShot      1        ''           timer-3
   2       singleShot      1        ''           timer-4
   3       singleShot      1        ''           timer-5
```

Using `timerfind` to determine all the timer objects that exist in memory can be helpful when deleting timer objects.

Finding Invisible Timer Objects

If you set the value of a timer object's `ObjectVisibility` property to `'off'`, the timer object does not appear in listings of existing timer objects returned by `timerfind`. The `ObjectVisibility` property provides a way for application developers to prevent end-user access to the timer objects created by their application.

Objects that are not visible are still valid. If you have access to the object (for example, from within the M-file that created it), you can set its properties. To

retrieve a list of all the timer objects in memory, including invisible ones, use the `timerfindall` function.

10

Performance

Analyzing Your Program's Performance

In this section...

Overview

The M-file Profiler graphical user interface and the stopwatch timer functions enable you to get back information on how your program is performing and help you identify areas that need improvement. The Profiler can be more useful in measuring relative execution time and in identifying specific performance bottlenecks in your code, while the stopwatch functions tend to be more useful for providing absolute time measurements.

The M-File Profiler Utility

A good first step to speeding up your programs is to find out where the bottlenecks are. This is where you need to concentrate your attention to optimize your code.

The MATLAB software provides the M-file Profiler, a graphical user interface that shows you where your program is spending its time during execution. Use the Profiler to help you determine where you can modify your code to make performance improvements.

To start the Profiler, type `profile viewer` or select **Desktop > Profiler** in the MATLAB Command Window. See Profiling for Improving Performance in the MATLAB Desktop Tools and Development Environment documentation, and the `profile` function reference page.

Stopwatch Timer Functions

If you just need to get an idea of how long your program (or a portion of it) takes to run, or to compare the speed of different implementations of a program, you can use the stopwatch timer functions, `tic` and `toc`. Invoking

`tic` starts the timer, and the first subsequent `toc` stops it and reports the time elapsed between the two.

Use `tic` and `toc` as shown here:

```
tic
   -- run the program section to be timed --
toc
```

Keep in mind that `tic` and `toc` measure overall elapsed time. Make sure that no other applications are running in the background on your system that could affect the timing of your MATLAB programs.

Measuring Smaller Programs

Shorter programs sometimes run too fast to get useful data from `tic` and `toc`. When this is the case, try measuring the program running repeatedly in a loop, and then average to find the time for a single run:

```
tic
   for k = 1:100
      -- run the program --
   end
toc
```

Using tic and toc Versus the cputime Function

Although it is possible to measure performance using the `cputime` function, it is recommended that you use the `tic` and `toc` functions for this purpose exclusively. It has been the general rule for CPU-intensive calculations run on Microsoft Windows machines that the elapsed time using `cputime` and the elapsed time using `tic` and `toc` are close in value, ignoring any first time costs. There are cases however that show a significant difference between these two methods. For example, in the case of a Pentium 4 with hyperthreading running Windows, there can be a significant difference between the values returned by `cputime` versus `tic` and `toc`.

Techniques for Improving Performance

In this section...

Vectorizing Loops

The MATLAB software uses a matrix language, which means it is designed for vector and matrix operations. You can often speed up your M-file code by using vectorizing algorithms that take advantage of this design. *Vectorization* means converting `for` and `while` loops to equivalent vector or matrix operations.

Simple Example of Vectorizing

Here is one way to compute the sine of 1001 values ranging from 0 to 10:

```
i = 0;
for t = 0:.01:10
    i = i + 1;
```

```
    y(i) = sin(t);
end
```

A vectorized version of the same code is

```
t = 0:.01:10;
y = sin(t);
```

The second example executes much faster than the first and is the way MATLAB is meant to be used. Test this on your system by creating M-file scripts that contain the code shown, and then using the `tic` and `toc` functions to time the M-files.

Advanced Example of Vectorizing

`repmat` is an example of a function that takes advantage of vectorization. It accepts three input arguments: an array `A`, a row dimension `M`, and a column dimension `N`.

`repmat` creates an output array that contains the elements of array `A`, replicated and “tiled” in an `M`-by-`N` arrangement:

```
A = [1 2 3; 4 5 6];

B = repmat(A,2,3);
B =
    1     2     3     1     2     3     1     2     3
    4     5     6     4     5     6     4     5     6
    1     2     3     1     2     3     1     2     3
    4     5     6     4     5     6     4     5     6
```

`repmat` uses vectorization to create the indices that place elements in the output array:

```
function B = repmat(A, M, N)

% Step 1 Get row and column sizes
[m,n] = size(A);

% Step 2 Generate vectors of indices from 1 to row/column size
mind = (1:m)';
```

```
nind = (1:n)';

% Step 3 Create index matrices from vectors above
mind = mind(:,ones(1, M));
nind = nind(:,ones(1, N));

% Step 4 Create output array
B = A(mind,nind);
```

Step 1, above, obtains the row and column sizes of the input array.

Step 2 creates two column vectors. `mind` contains the integers from 1 through the row size of `A`. The `nind` variable contains the integers from 1 through the column size of `A`.

Step 3 uses a MATLAB vectorization trick to replicate a single column of data through any number of columns. The code is

```
B = A(:,ones(1,nCols))
```

where `nCols` is the desired number of columns in the resulting matrix.

Step 4 uses array indexing to create the output array. Each element of the row index array, `mind`, is paired with each element of the column index array, `nind`, using the following procedure:

1. The first element of `mind`, the row index, is paired with each element of `nind`. MATLAB moves through the `nind` matrix in a columnwise fashion, so `mind(1,1)` goes with `nind(1,1)`, and then `nind(2,1)`, and so on. The result fills the first row of the output array.

2. Moving columnwise through `mind`, each element is paired with the elements of `nind` as above. Each complete pass through the `nind` matrix fills one row of the output array.

Caution While `repmat` can take advantage of vectorization, it can do so at the expense of memory usage. When this is the case, you might find the `bsxfun` function be more appropriate in this respect.

Functions Used in Vectorizing

Some of the most commonly used functions for vectorizing are as follows

Function	Description
`all`	Test to determine if all elements are nonzero
`any`	Test for any nonzeros
`cumsum`	Find cumulative sum
`diff`	Find differences and approximate derivatives
`find`	Find indices and values of nonzero elements
`ind2sub`	Convert from linear index to subscripts
`ipermute`	Inverse permute dimensions of a multidimensional array
`logical`	Convert numeric values to logical
`ndgrid`	Generate arrays for multidimensional functions and interpolation
`permute`	Rearrange dimensions of a multidimensional array
`prod`	Find product of array elements
`repmat`	Replicate and tile an array
`reshape`	Change the shape of an array
`shiftdim`	Shift array dimensions
`sort`	Sort array elements in ascending or descending order
`squeeze`	Remove singleton dimensions from an array
`sub2ind`	Convert from subscripts to linear index
`sum`	Find the sum of array elements

Preallocating Arrays

`for` and `while` loops that incrementally increase, or *grow*, the size of a data structure each time through the loop can adversely affect performance and memory use. Repeatedly resizing arrays often requires that MATLAB spend extra time looking for larger contiguous blocks of memory and then moving the array into those blocks. You can often improve on code execution time by

preallocating the maximum amount of space that would be required for the array ahead of time.

The following code creates a scalar variable `x`, and then gradually increases the size of `x` in a `for` loop instead of preallocating the required amount of memory at the start:

```
x = 0;
for k = 2:1000
   x(k) = x(k-1) + 5;
end
```

Change the first line to preallocate a 1-by-1000 block of memory for `x` initialized to zero. This time there is no need to repeatedly reallocate memory and move data as more values are assigned to `x` in the loop:

```
x = zeros(1, 1000);
for k = 2:1000
   x(k) = x(k-1) + 5;
end
```

Preallocation Functions

Preallocation makes it unnecessary for MATLAB to resize an array each time you enlarge it. Use the appropriate preallocation function for the kind of array you are working with.

Array Type	Function	Examples
Numeric	`zeros`	`y = zeros(1, 100);`

Cell	`cell`	`B = cell(2, 3);`
		`B{1,3} = 1:3;`
		`B{2,2} = 'string';`

Preallocating a Nondouble Matrix

When you preallocate a block of memory to hold a matrix of some type other than `double`, avoid using the method

```
A = int8(zeros(100));
```

This statement preallocates a 100-by-100 matrix of `int8` first by creating a full matrix of `doubles`, and then converting each element to `int8`. This costs time and uses memory unnecessarily.

The next statement shows how to do this more efficiently:

```
A = zeros(100, 'int8');
```

Use Distributed Arrays for Large Datasets

This topic is described in the “Parallel Math” section of the Parallel Computing Toolbox™ documentation.

When Possible, Replace for with parfor (Parallel for)

This topic is described in the “Parallel for-Loops” section of the Parallel Computing Toolbox documentation.

Multithreading Capabilities in MATLAB

See “Implicit Multiprocessing” on page 10-14 to learn more about making use of multithreaded computation.

Limiting M-File Size and Complexity

Running programs that are unusually large or complex can put a strain on your system’s resources. For example, a program that nearly exceeds memory capacity may work some of the time and sometimes not, depending on the commands it uses and on what other applications are running at the time. An example of unnecessary complexity might be having a large number of `if` and `else` statements where `switch` and `case` might be more suitable. This can also lead to performance and space problems. If you see the following error message displayed, this is likely to be the source of the problem:

```
The input was too complicated or too big for MATLAB to parse
```

If you have an M-file that includes thousands of variables or functions, tens of thousands of statements, or hundreds of language keyword pairs (e.g., `if-else`, or `try-catch`), then making some of the changes suggested here is likely to not only boost its performance and reliability, but should make your program code easier to understand and maintain as well.

- Split large script files into smaller ones, having the first file call the second if necessary.
- Take your larger chunks of program code and make separate functions (or subfunctions and nested functions) of them.
- If you have functions or expressions by that seem overly complicated, make smaller and simpler functions or expressions of them. Simpler functions are also more likely to be made into utility functions that you can share with others.

Coding Loops in a MEX-File

If there are instances where you cannot vectorize and must use a `for` or `while` loop, consider coding the loop in a MEX-file. In this way, the loop executes much more quickly since the instructions in the loop do not have to be interpreted each time they execute.

See "Using MEX-Files to Call C and Fortran Programs" in the External Interfaces documentation.

Assigning to Variables

For best performance, keep the following suggestions in mind when assigning values to variables.

Changing a Variable's Data Type or Dimension

Changing the class or array shape of an existing variable slows MATLAB down as it must take extra time to process this. When you need to store data of a different type, it is advisable to create a new variable.

This code changes the type for `X` from `double` to `char`, which has a negative impact on performance:

```
X = 23;
     .
-- other code  --
     .
X = 'A';                    % X changed from type double to char
     .
-- other code  --
```

Assigning Real and Complex Numbers

Assigning a complex number to a variable that already holds a real number impacts the performance of your program. Similarly, you should not assign a real value to a variable that already holds a complex value.

Operating on Real Data

When operating on real (i.e., noncomplex) numbers, it is more efficient to use MATLAB functions that have been designed specifically for real numbers. The following functions return numeric values that are real.

Function	Description
`reallog`	Find natural logarithm for nonnegative real arrays
`realpow`	Find array power for real-only output
`realsqrt`	Find square root for nonnegative real arrays

Using Appropriate Logical Operators

When performing a logical AND or OR operation, you have a choice of two operators of each type.

Operator	Description
`&`, `|`	Perform logical AND and OR on arrays element by element
`&&`, `||`	Perform logical AND and OR on scalar values with short-circuiting

In `if` and `while` statements, it is more efficient to use the short-circuiting operators, `&&` for logical AND and `||` for logical OR. This is because these operators often do not have to evaluate the entire logical expression. For example, MATLAB evaluates only the first part of this expression whenever the number of input arguments is less than three:

```
if (nargin >= 3) && (ischar(varargin{3}))
```

See Short-Circuit Operators in the MATLAB documentation for a discussion on short-circuiting with `&&` and `||`.

Overloading Built-In Functions

Overloading MATLAB built-in functions on any of the standard MATLAB data classes can negatively affect performance. For example, if you overload the `plus` function to handle any of the integer classes differently, you may hinder certain optimizations in the MATLAB built-in function code for `plus`, and thus may slow down any programs that make use of this overload.

Functions Are Generally Faster Than Scripts

Your code executes more quickly if it is implemented in a function rather than a script.

Load and Save Are Faster Than File I/O Functions

If you have a choice of whether to use `load` and `save` instead of the low-level MATLAB file I/O routines such as `fread` and `fwrite`, choose the former. `load` and `save` have been optimized to run faster and reduce memory fragmentation.

Avoid Large Background Processes

Avoid running large processes in the background at the same time you are executing your program in MATLAB. This frees more CPU time for your MATLAB session.

MATLAB Multiprocessing

In this section...

Overview

The MATLAB software supports two types of multiprocessing: *implicit* and *explicit*.

Implicit Multiprocessing

Characteristics of implicit multiprocessing:

- Runs multiple threads on a single machine, most often using one thread per processing unit.
- Requires a multiple CPU (multiprocessor or multicore) system.
- Speeds up elementwise computations such as those done by the `sin` and `log` functions, and computations that use the Basic Linear Algebra Subroutines (BLAS) library, such as matrix multiply.
- Does *not* require any changes to your MATLAB code.
- Works behind the scenes to take advantage of the processing units available to you. It does this by multithreading the computationally-intensive math library functions that you use in the course of your MATLAB session.

MATLAB enables multithreaded computation by default. You can disable it by specifying the `singleCompThread` option when starting MATLAB.

Explicit Multiprocessing

Characteristics of explicit multiprocessing:

- Runs separate processes on one or many machines.
- Requires installation of Parallel Computing Toolbox.

- Speeds up execution of large MATLAB jobs. Enables you to run jobs simultaneously on a cluster of computers, or as several processes on a single machine.
- Requires that you modify your MATLAB code.
- The Parallel Computing Toolbox supports programming constructs for distributed arrays and parallel `for` (`parfor`) loops. It also supports both interactive and batch execution.

Enable explicit multiprocessing by installing Parallel Computing Toolbox.

Implicit Multiprocessing

Multithreaded computation runs in a single instance of MATLAB and generates simultaneous instruction streams on a multiple CPU (multiprocessor or multicore) system. The multiple processors share the memory of a single computer. The work to be processed is implicitly partitioned for execution on multiple threads. Multithreaded computation in MATLAB speeds up elementwise computations such as those done by the `sin` and `log` functions, and computations that use the Basic Linear Algebra Subroutines (BLAS) library, such as matrix multiply.

If you are using a multiple-CPU system, you can run a demo to see the performance impact—see Multithreaded Computation in the Help browser **Demos** pane, under MATLAB Mathematics.

For information regarding specific functions, search for “Which MATLAB Functions Support Multithreaded Computation” on The MathWorks online Support page.

Platform Differences and Multithreaded Computation

The BLAS library used for multithreaded computation differs according to which platform you are using:

Platform	BLAS Used
Windows with Intel® processors	Intel MKL BLAS
Windows with AMD processors	AMD® ACML™ BLAS

Platform	BLAS Used
Linux[1] with Intel processors	Intel MKL BLAS
Linux with AMD processors	AMD ACML BLAS
Macintosh Intel-based	Intel MKL BLAS
Solaris™	Sun Performance Library BLAS

Enabling Multithreaded Computation

Note See "MATLAB Multiprocessing" on page 10-13 for an overview of the multiprocessing capabilities provided with MATLAB.

Multithreaded computation in MATLAB is enabled by default. When enabled, MATLAB automatically detects the number of CPUs on your system and recommends the number of threads based on that. To disable multithreaded computation, start MATLAB using the `singleCompThread` option.

To set or retrieve the maximum number of computational threads from the command line or from within an M-file program, use the `maxNumCompThreads` function. You can either set the maximum number of computational threads to a specific number, or indicate that you want the setting to be done automatically by MATLAB.

To set the maximum number of computational threads to a specific number `N`, use

```
maxNumCompThreads(N)
```

To have MATLAB set the maximum number of threads, use:

```
maxNumCompThreads('automatic')
```

`maxNumCompThreads` also returns the current maximum number of threads if you call it with an output value:

```
oldN = MaxNumCompThreads(newN)
```

1. Linux is a registered trademark of Linus Torvalds.

Setting the Number of Threads Programmatically

Explicit Multiprocessing

See the Parallel Computing Toolbox documentation for information regarding explicit multiprocessing in MATLAB.

11

Memory Usage

Memory Allocation

In this section...
“Memory Allocation for Arrays” on page 11-2
“Data Structures and Memory” on page 11-7

For more information on memory management, see Technical Note 1106: “Memory Management Guide” at the following URL:

`http://www.mathworks.com/support/tech-notes/1100/1106.html`

Memory Allocation for Arrays

The topics below provide information on how the MATLAB software allocates memory when working with arrays and variables. The purpose is to help you use memory more efficiently when writing code. Most of the time, however, you should not need to be concerned with these internal operations as MATLAB handles data storage for you automatically.

Note Any information on how the MATLAB software handles data internally is subject to change in future releases.

Creating and Modifying Arrays

When you assign a numeric or character array to a variable, MATLAB allocates a contiguous virtual block of memory and stores the array data in that block. MATLAB also stores information about the array data, such as its class and dimensions, in a separate, small block of memory called a *header*.

If you add new elements to an existing array, MATLAB expands the existing array in memory in a way that keeps its storage contiguous. This usually requires finding a new block of memory large enough to hold the expanded array. MATLAB then copies the contents of the array from its original location to this new block in memory, adds the new elements to the array in this block, and frees up the original array location in memory.

If you remove elements from an existing array, MATLAB keeps the memory storage contiguous by removing the deleted elements, and then compacting its storage in the original memory location.

Working with Large Data Sets. If you are working with large data sets, you need to be careful when increasing the size of an array to avoid getting errors caused by insufficient memory. If you expand the array beyond the available contiguous memory of its original location, MATLAB must make a copy of the array and set this copy to the new value. During this operation, there are two copies of the original array in memory. This temporarily doubles the amount of memory required for the array and increases the risk of your program running out of memory during execution. It is better to preallocate sufficient memory for the largest potential size of the array at the start. See “Preallocating Arrays” on page 10-7.

Copying Arrays

Internally, multiple variables can point to the same block of data, thus sharing that array’s value. When you copy a variable to another variable (e.g., `B = A`), MATLAB makes a copy of the array reference, but not the array itself. As long as you do not modify the contents of the array, there is no need to store more than one copy of it. If you do modify any elements of the array, MATLAB makes a copy of the array and then modifies that copy.

The following example demonstrates this. Start by creating a simple M-file script `memUsed.m` to display how much memory is currently being used by your MATLAB process. Put these two lines of code in the script:

```
[usr, sys] = memory;
usr.MemUsedMATLAB
```

Get an initial reading of how much memory is currently being used by your MATLAB process:

```
format short eng;
memUsed
ans =
   295.4977e+006
```

Create a 2000-by-2000 numeric array A. This uses about 32MB of memory:

```
A = magic(2000);
memUsed
ans =
   327.6349e+006
```

Make a copy of array `A` in `B`. As there is no need at this point to have two copies of the array data, MATLAB only makes a copy of the array reference. This requires no significant additional memory:

```
B = A;
memUsed
ans =
   327.6349e+006
```

Now modify `B` by making it one half its original size (i.e., set 1000 rows to empty). This requires that MATLAB make a copy of at least the first 1000 rows of the `A` array, and assign that copy to `B`:

```
B(1001:2000,:) = [];
format short;   size(B)
ans =
        1000        2000
```

Check the memory used again. Even though `B` is significantly smaller than it was originally, the amount of memory used by the MATLAB process has increased by about 16 MB (1/2 of the 32 MB originally required for `A`) because B could no longer remain as just a reference to `A`:

```
format short eng;   memUsed
ans =
   343.6421e+006
```

Array Headers

When you assign an array to a variable, MATLAB also stores information about the array (such as class and dimensions) in a separate piece of memory called a header. For most arrays, the memory required to store the header is insignificant. There is a small advantage to storing large data sets in a small number of large arrays as opposed to a large number of small arrays. This is because the former configuration requires fewer array headers.

Structure and Cell Arrays. For structures and cell arrays, MATLAB creates a header not only for each array, but also for each field of the structure and for each cell of a cell array. Because of this, the amount of memory required to store a structure or cell array depends not only on how much data it holds, but also on how it is constructed.

For example, take a scalar structure array `S1` having fields `R`, `G`, and `B`. Each field of size 100-by-50 requires one array header to describe the overall structure, one header for each unique field name, and one header per field for the 1-by-1 structure array. This makes a total of seven array headers for the entire data structure:

```
S1.R(1:100,1:50)
S1.G(1:100,1:50)
S1.B(1:100,1:50)
```

On the other hand, take a 100-by-50 structure array `S2` in which each element has scalar fields `R`, `G`, and `B`. You only need one array header to describe the overall structure, one for each unique field name, and one per field for each of the 5,000 elements of the structure, making a total of 15,004 array headers for the entire data structure:

```
S2(1:100,1:50).R
S2(1:100,1:50).G
S2(1:100,1:50).B
```

Even though `S1` and `S2` contain the same amount of data, `S1` uses significantly less space in memory. Not only is less memory required, but there is a corresponding speed benefit to using the `S1` format, as well.

See “Cell Arrays” and “Structures” under “Data Structures and Memory” on page 11-7.

Memory Usage Reported By the whos Function. The whos function displays the amount of memory consumed by any variable. For reasons of simplicity, whos reports only the memory used to store the actual data. It does not report storage for the array header, for example.

Function Arguments

MATLAB handles arguments passed in function calls in a similar way. When you pass a variable to a function, you are actually passing a reference to the data that the variable represents. As long as the input data is not modified by the function being called, the variable in the calling function and the variable in the called function point to the same location in memory. If the called function modifies the value of the input data, then MATLAB makes a copy of the original array in a new location in memory, updates that copy with the modified value, and points the input variable in the called function to this new array.

In the example below, function myfun modifies the value of the array passed into it. MATLAB makes a copy in memory of the array pointed to by A, sets variable X as a reference to this new array, and then sets one row of X to zero. The array referenced by A remains unchanged:

```
A = magic(500);
myfun(A);

function myfun(X)
X(400,:) = 0;
```

If the calling function needs the modified value of the array it passed to myfun, you need to return the updated array as an output of the called function, as shown here for variable A:

```
A = magic(500);
A = myfun(A);
sprintf('The new value of A is %d', A)

function Y = myfun(X)
X(400,:) = 0;
Y = X;
```

Data Structures and Memory

Memory requirements differ for the various types of MATLAB data structures. You may be able to reduce the amount of memory used for these structures by considering how MATLAB stores them.

Numeric Arrays

MATLAB requires 1, 2, 4, or 8 bytes to store 8-bit, 16-bit, 32-bit, and 64-bit signed and unsigned integers, respectively. For floating-point numbers, MATLAB uses 4 or 8 bytes for `single` and `double` types. To conserve memory when working with numeric arrays, The MathWorks recommends that you use the smallest integer or floating-point type that will contain your data without overflowing. For more information, see "Numeric Types" in the MATLAB Programming Fundamentals documentation.

Complex Arrays

MATLAB stores complex data as separate real and imaginary parts. If you make a copy of a complex array variable, and then modify only the real or imaginary part of the array, MATLAB creates a new array containing both real and imaginary parts.

Sparse Matrices

It is best to store matrices with values that are mostly zero in sparse format. Sparse matrices can use less memory and may also be faster to manipulate than full matrices. You can convert a full matrix to sparse format using the `sparse` function.

Compare two 1000-by-1000 matrices: `X`, a matrix of doubles with 2/3 of its elements equal to zero; and `Y`, a sparse copy of `X`. The following example shows that the sparse matrix requires approximately half as much memory:

```
whos
  Name      Size                     Bytes  Class

  X      1000x1000                 8000000  double array
  Y      1000x1000                 4004000  double array (sparse)
```

Cell Arrays

In addition to data storage, cell arrays require a certain amount of additional memory to store information describing each cell. This information is recorded in a *header*, and there is one header for each cell of the array. You can determine the amount of memory required for a cell array header by finding the number of bytes consumed by a 1-by-1 cell that contains no data, as shown below for a 32-bit system:

```
A = {[]};      % Empty cell array

whos A
  Name      Size            Bytes  Class    Attributes

  A         1x1                60  cell
```

In this case, MATLAB shows the number of bytes required for each header in the cell array on a 32-bit system to be 60. This is the header size that is used in all of the 32-bit examples in this section. For 64-bit systems, the header size is assumed to be 112 bytes in this documentation. You can find the correct header size on a 64-bit system using the method just shown for 32 bits.

To predict the size of an entire cell array, multiply the number you have just derived for the header by the total number of cells in the array, and then add to that the number of bytes required for the data you intend to store in the array:

```
(header_size x number_of_cells) + data
```

So a 10-by-20 cell array that contains 400 bytes of data would require 22,800 bytes of memory on a 64-bit system:

```
(112 x 200) + 400 = 22800
```

Note While numeric arrays must be stored in contiguous memory, structures and cell arrays do not.

Example 1 – Memory Allocation for a Cell Array. The following 4-by-1 cell array records the brand name, screen size, price, and on-sale status for three laptop computers:

```
Laptops = {['SuperrrFast 89X', 'ReliablePlus G5', ...
            'UCanA4dIt 140L6']; ...
           [single(17), single(15.4), single(14.1)]; ...
           [2499.99, 1199.99, 499.99]; ...
           [true, true, false]};
```

On a 32-bit system, the cell array header alone requires 60 bytes per cell:

```
4 cells * 60 bytes per cell = 240 bytes for the cell array
```

Calculate the memory required to contain the data in each of the four cells:

```
45 characters * 2 bytes per char = 90 bytes
 3 doubles * 8 bytes per double = 24 bytes
 3 singles * 4 bytes per single = 12 bytes
 3 logicals * 1 byte per logical = 3 bytes

90 + 24 + 12 + 3 = 129 bytes for the data
```

Add the two, and then compare your result with the size returned by MATLAB:

```
240 + 129 = 369 bytes total

whos Laptops
  Name         Size             Bytes  Class    Attributes

  Laptops      4x1                369  cell
```

Structures

```
S.A = [];
B = whos('S');
B.bytes - 60
ans =
    64
```

Compute the memory needed for a structure array as follows:

```
32-bit systems:  fields x ((60 x array elements) + 64) + data
64-bit systems:  fields x ((112 x array elements) + 64) + data
```

On a 64-bit computer system, a 4-by-5 structure `Clients` with fields `Address` and `Phone` uses 4,608 bytes just for the structure:

```
2 fields x ((112 x 20) + 64) = 2 x (2240 + 64) = 4608 bytes
```

To that sum, you must add the memory required to hold the data assigned to each field. If you assign a 25-character string to `Address` and a 12-character string to `Phone` in each element of the 4-by-5 `Clients` array, you use 1480 bytes for data:

```
(25+12) characters * 2 bytes per char * 20 elements = 1480 bytes
```

Add the two and you see that the entire structure consumes 6,088 bytes of memory.

Example 1 – Memory Allocation for a Structure Array. Compute the amount of memory that would be required to store the following 6-by-5 structure array having the following four fields on a 32-bit system:

```
A:  5-by-8-by-6 signed 8-bit integer array
B:  1-by-200 single array
C:  30-by-30 unsigned 16-bit integer array
D:  1-by-27 character array
```

Construct the array:

```
A = int8(ones(5,8,6));
B = single(1:500);
C = uint16(magic(30));
D = 'Company Name: The MathWorks';

s = struct('f1', A, 'f2', B, 'f3', C, 'f4', D);

for m=1:6
   for n=1:5
      s(m,n)=s(1,1);
   end
end
```

Calculate the amount of memory required for the structure itself, and then for the data it contains:

```
structure = fields x ((60 x array elements) + 64) =
            4 x ((60 x 30) + 64) = 7456 bytes

data = (field1 + field2 + field3 + field4) x array elements =
       (240 + 2000 + 1800 + 54) x 30 = 122,820 bytes
```

Add the two, and then compare your result with the size returned by MATLAB:

```
Total bytes calculated for structure s: 7456 + 122,820 = 130,276

whos s
  Name      Size              Bytes  Class     Attributes

  s         6x5               130276  struct
```

Memory Management Functions

The following functions can help you to manage memory use while running the MATLAB software:

- `memory` displays or returns information about how much memory is available and how much is used by MATLAB. This includes the following:
  - Size of the largest single array MATLAB can create at this time.
  - Total size of the virtual address space available for data.
  - Total amount of memory used by the MATLAB process for both libraries and data.
  - Available and total Virtual Memory for the MATLAB software process.
  - Available system memory, including both physical memory and paging file.
  - Available and the total physical memory (RAM) of the computer.
- `whos` shows how much memory MATLAB currently has allocated for variables in the workspace.
- `pack` saves existing variables to disk, and then reloads them contiguously. This reduces the chances of running into problems due to memory fragmentation.
- `clear` removes variables from memory. One way to increase the amount of available memory is to periodically clear variables from memory that you no longer need.

  If you use `pack` and there is still not enough free memory to proceed, you probably need to remove some of the variables you are no longer using from memory. Use `clear` to do this.
- `save` selectively stores variables to the disk. This is a useful technique when you are working with large amounts of data. Save data to the disk periodically, and then use the `clear` function to remove the saved data from memory.
- `load` reloads a data file saved with the `save` function.
- `quit` exits MATLAB and returns all allocated memory to the system. This can be useful on The Open Group UNIX systems, which do not free up

memory allocated to an application (e.g., MATLAB) until the application exits.

You can use the `save` and `load` functions in conjunction with the `quit` command to free memory by:

1. Saving any needed variables with the `save` function.

2. Quitting MATLAB to free all memory allocated to MATLAB.

3. Starting a new MATLAB session and loading the saved variables back into the clean MATLAB workspace.

The whos Function

The whos command can give you an idea of the memory used by MATLAB variables.

```
A = ones(10,10);
whos
   Name       Size            Bytes  Class     Attributes
   A         10x10              800  double
```

Note that whos does not include information about

- Memory used by MATLAB (e.g., Sun Java code and plots).
- Memory used for most objects (e.g., time series, custom) .
- Memory for variables not in the calling workspace .
- Shared data copies (e.g., memory for B in >>B=A;) will still have bytes used listed against it even when it does not use any memory (see the "Be aware of the Function Argument Passing Model" section earlier).

Strategies for Efficient Use of Memory

In this section...

Ways to Reduce the Amount of Memory Required

The source of many "out of memory" problems often involves analyzing or processing an existing large set of data such as in a file or a database. This requires bringing all or part of the data set into the MATLAB software process. The following techniques deal with minimizing the required memory during this stage.

Load Only As Much Data As You Need

Only import into MATLAB as much of a large data set as you need for the problem you are trying to solve. This is not usually a problem when importing from sources, such as a database where you can explicitly search for elements matching a query. But this is a common problem with loading large flat text or binary files. Many users are tempted to try and load the entire file first, and then process it with MATLAB. Be sure to use the appropriate MATLAB function to load parts of files.

Text Files. Use the `textscan` function to access parts of a large text file by reading only the selected columns and rows. If you specify the number of rows or a repeat format number with `textscan`, MATLAB calculates the exact amount of memory required beforehand.

Binary Files. You can use low-level binary file I/O functions, such as `fread`, to access parts of any file that has a known format. For binary files of an unknown format, try using memory mapping with the `memmapfile` function.

MAT-Files. Use the `whos` function with the `-file` option to preview the file. This command displays each array in the MAT-file that you specify and the number of bytes in the array:

```
whos -file session1.mat
  Name        Size              Bytes  Class     Attributes

  S2          1x1                 723  struct
  x           100x200              72  double    sparse
  Mat4        4x20                640  double
  A           3151872x1       3151872  uint8
  Seq         1x912211         912211  int8
```

If there are large arrays in the MAT-file that you do not need for your current task, you can selectively import only those variables that you want using `load`.

HDF Files. You can load parts of HDF files using the `hdfread` and `hdf5read` functions.

Image, Audio, and Video Files. The MATLAB functions that support loading from these types of files usually require that you import the entire file. To import portions of the file, use low-level I/O functions such as `fread`.

Process Data By Blocks

Consider block processing, that is, processing a large data set one section at a time in a loop. Reducing the size of the largest array in a data set reduces the size of any copies or temporaries needed. You can use this technique in either of two ways:

- For a subset of applications that you can break into separate chunks and process independently.
- For applications that only rely on the state of a previous block, such as filtering.

Avoid Creating Temporary Arrays

Avoid creating large temporary variables, and also make it a practice to clear those temporary variables you do use when they are no longer needed. For example, when you create a large array of zeros, instead of saving to a temporary variable `A`, and then converting `A` to a single:

```
A = zeros(1e6,1);
As = single(A);
```

use just the one command to do both operations:

```
A = zeros(1e6,1,'single');
```

Using the `repmat` function, array preallocation and `for` loops are other ways to work on `nondouble` data without requiring temporary storage in memory.

Use Nested Functions to Pass Fewer Arguments

When working with large data sets, be aware that MATLAB makes a temporary copy of an input variable if the called function modifies its value. This temporarily doubles the memory required to store the array, which causes MATLAB to generate an error if sufficient memory is not available.

One way to use less memory in this situation is to use nested functions. A nested function shares the workspace of all outer functions, giving the nested function access to data outside of its usual scope. In the example shown here, nested function `setrowval` has direct access to the workspace of the outer function `myfun`, making it unnecessary to pass a copy of the variable in the function call. When `setrowval` modifies the value of `A`, it modifies it in the workspace of the calling function. There is no need to use additional memory to hold a separate array for the function being called, and there also is no need to return the modified value of `A`:

```
function myfun
A = magic(500);

   function setrowval(row, value)
   A(row,:) = value;
   end

setrowval(400, 0);
disp('The new value of A(399:401,1:10) is')
A(399:401,1:10)
end
```

Using Appropriate Data Storage

MATLAB provides you with different sizes of data classes, such as `double` and `uint8`, so you do not need to use large classes to store your smaller segments

of data. For example, it takes 7,000 KB less memory to store 1,000 small unsigned integer values using the `uint8` class than it does with `double`.

Use the Appropriate Numeric Class

The numeric class you should use in MATLAB depends on your intended actions. The default class `double` gives the best precision, but requires 8 bytes per element of memory to store. If you intend to perform complicated math such as linear algebra, you must use a floating-point class such as a `double` or `single`. The `single` class requires only 4 bytes. There are some limitations on what you can do with `singles`, but most MATLAB Math operations are supported.

If you just need to carry out simple arithmetic and you represent the original data as integers, you can use the integer classes in MATLAB. The following is a list of numeric classes, memory requirements (in bytes), and the supported operations.

Class (Data Type)	Bytes	Supported Operations
`single`	4	Most math
`double`	8	All math
`logical`	1	Logical/conditional operations
`int8, uint8`	1	Arithmetic and some simple functions
`int16, uint16`	2	Arithmetic and some simple functions
`int32, uint32`	4	Arithmetic and some simple functions
`int64, int64`	8	Arithmetic and some simple functions

Reduce the Amount of Overhead When Storing Data

MATLAB arrays (implemented internally as `mxArrays`) require room to store meta information about the data in memory, such as type, dimensions, and attributes. This takes about 80 bytes per array. This overhead only becomes an issue when you have a large number (e.g., hundreds or thousands) of small `mxArrays` (e.g., scalars). The `whos` command lists the memory used by variables, but does not include this overhead.

Because simple numeric arrays (comprising one mxArray) have the least overhead, you should use them wherever possible. When data is too complex to store in a simple array (or matrix), you can use other data structures.

Cell arrays are comprised of separate mxArrays for each element. As a result, cell arrays with many small elements have a large overhead.

Structures require a similar amount of overhead per field (see the documentation on "Array Headers" on page 11-5 above). Structures with many fields and small contents have a large overhead and should be avoided. A large array of structures with numeric scalar fields requires much more memory than a structure with fields containing large numeric arrays.

Also note that while MATLAB stores numeric arrays in contiguous memory, this is not the case for structures and cell arrays.

Import Data to the Appropriate MATLAB Class

When reading data from a binary file with fread, it is a common error to specify only the class of the data in the file, and not the class of the data MATLAB uses once it is in the workspace. As a result, the default double is used even if you are reading only 8-bit values. For example,

```
fid = fopen('large_file_of_uint8s.bin', 'r');
a = fread(fid, 1e3, 'uint8');            % Requires 8k
whos a
  Name         Size            Bytes  Class    Attributes

  a         1000x1              8000  double

a = fread(fid, 1e3, 'uint8=>uint8');      % Requires 1k
whos a
  Name         Size            Bytes  Class    Attributes

  a         1000x1              1000  uint8
```

Make Arrays Sparse When Possible

If your data contains many zeros, consider using sparse arrays, which store only nonzero elements. The example below compares the space required for storage of an array of mainly zeros:

```
A = diag(1e3,1e3);    % Full matrix with ones on the diagonal
As = sparse(A)        % Sparse matrix with only nonzero elements
whos
  Name      Size                    Bytes  Class

  A       1001x1001                 8016008  double array
  As      1001x1001                 4020     double array (sparse)
```

You can see that this array requires only approximately 4 KB to be stored as sparse, but approximately 8 MB as a full matrix. In general, for a sparse double array with `nnz` nonzero elements and `ncol` columns, the memory required is

- 16 * `nnz` + 8 * `ncol` + 8 bytes (on a 64 bit machine)
- 12 * `nnz` + 4 * `ncol` + 4 bytes (on a 32 bit machine)

Note that MATLAB does not support all mathematical operations on sparse arrays.

How to Avoid Fragmenting Memory

MATLAB always uses a contiguous segment of memory to store a numeric array. As you manipulate this data, however, the contiguous block can become fragmented. When memory is fragmented, there may be plenty of free space, but not enough contiguous memory to store a new large variable. Increasing fragmentation can use significantly more memory than is necessary.

Preallocate Contiguous Memory When Creating Arrays

In the course of a MATLAB session, memory can become fragmented due to dynamic memory allocation and deallocation. `for` and `while` loops that incrementally increase, or *grow*, the size of a data structure each time through the loop can add to this fragmentation as they have to repeatedly find and allocate larger blocks of memory to store the data.

To make more efficient use of your memory, preallocate a block of memory large enough to hold the matrix at its final size before entering the loop. When you preallocate memory for an array, MATLAB reserves sufficient contiguous space for the entire full-size array at the beginning of the computation. Once

you have this space, you can add elements to the array without having to continually allocate new space for it in memory.

For more information on preallocation, see "Preallocating Arrays" on page 10-7.

Allocate Your Larger Arrays First

MATLAB uses a heap method of memory management. It requests memory from the operating system when there is not enough memory available in the heap to store the current variables. It reuses memory as long as the size of the memory segment required is available in the heap.

The following statements can require approximately 4.3 MB of RAM. This is because MATLAB may not be able to reuse the space previously occupied by two 1 MB arrays when allocating space for a 2.3 MB array:

```
a = rand(1e6,1);
b = rand(1e6,1);
clear
c = rand(2.3e6,1);
```

The simplest way to prevent overallocation of memory is to allocate the largest vectors first. These statements require only about 2.0 MB of RAM:

```
c = rand(2.3e6,1);
clear
a = rand(1e6,1);
b = rand(1e6,1);
```

Long-Term Usage (Windows Systems Only)

On 32-bit Microsoft Windows, the workspace of MATLAB can fragment over time due to the fact that the Windows memory manager does not return blocks of certain types and sizes to the operating system. Clearing the MATLAB workspace does not fix this problem. You can minimize the problem by allocating the largest variables first. This cannot address, however, the eventual fragmentation of the workspace that occurs from continual use of MATLAB over many days and weeks, for example. The only solution to this is to save your work and restart MATLAB.

The `pack` command, which saves all variables to disk and loads them back, does not help with this situation.

Reclaiming Used Memory

One simple way to increase the amount of memory you have available is to clear large arrays that you no longer use.

Save Your Large Data Periodically to Disk

If your program generates very large amounts of data, consider writing the data to disk periodically. After saving that portion of the data, use the `clear` function to remove the variable from memory and continue with the data generation.

Clear Old Variables from Memory When No Longer Needed

When you are working with a very large data set repeatedly or interactively, clear the old variable first to make space for the new variable. Otherwise, MATLAB requires temporary storage of equal size before overriding the variable. For example,

```
a = rand(100e6,1)              % 800 MB array
a = rand(100e6,1)              % New 800 MB array
??? Error using ==> rand
Out of memory. Type HELP MEMORY for your options.

clear a
a = rand(100e6,1)              % New 800 MB array
```

Resolving "Out of Memory" Errors

In this section...

General Suggestions for Reclaiming Memory

The MATLAB software generates an `Out of Memory` message whenever it requests a segment of memory from the operating system that is larger than what is currently available. When you see the `Out of Memory` message, use any of the techniques discussed under "Strategies for Efficient Use of Memory" on page 11-14 to help optimize the available memory. If the `Out of Memory` message still appears, you can try any of the following:

- Compress data to reduce memory fragmentation.
- If possible, break large matrices into several smaller matrices so that less memory is used at any one time.
- If possible, reduce the size of your data.
- Make sure that there are no external constraints on the memory accessible to MATLAB. (On The Open Group UNIX[2] systems, use the `limit` command to check).
- Increase the size of the swap file. We recommend that you configure your system with twice as much swap space as you have RAM. See "Increasing System Swap Space" on page 11-25, below.
- Add more memory to the system.

2. UNIX is a registered trademark of The Open Group in the United States and other countries.

Setting the Process Limit

The platforms and operating systems that MATLAB supports have different memory characteristics and limitations. In particular, the *process limit* is the maximum amount of virtual memory a single process (or application) can address. On 32-bit systems, this is the most important factor limiting data set size. The process limit must be large enough for MATLAB to store all of the data it is to process, plus M code, the MATLAB executable itself, and additional state information.

Where possible, choose an operating system that maximizes this number, that is, a 64-bit operating system. The following is a list of MATLAB supported operating systems and their process limits.

Operating System	Process Limit
32-bit Microsoft Windows XP, Windows Vista™.	2 GB
32-bit Windows XP with 3 GB `boot.ini` switch or 32-bit Windows Vista with `increaseuserva` set (see later)	3 GB
32-bit Linux (Linux is a registered trademark of Linus Torvalds.)	~3 GB
64-bit Windows XP, Apple® Macintosh OS X, Linux, or Sun Solaris running 32-bit MATLAB	≤ 4 GB
64-bit Windows XP, Windows Vista, Linux, or Solaris running 64-bit MATLAB	8 TB

To verify the current process limit of MATLAB on Windows systems, use the `memory` function.

```
Maximum possible array:             583 MB (6.111e+008 bytes) *
Memory available for all arrays:   1515 MB (1.588e+009 bytes) **
Memory used by MATLAB:              386 MB (4.050e+008 bytes)
Physical Memory (RAM):             2014 MB (2.112e+009 bytes)
```

```
*  Limited by contiguous virtual address space available.
** Limited by virtual address space available.
```

When called with one output variable, the `memory` function returns or displays the following values. See the function reference for `memory` to find out how to use it with more than one output.

memory Return Value	Description
`MaxPossibleArrayBytes`	Size of the largest single array MATLAB can currently create
`MemAvailableAllArrays`	Total size of the virtual address space available for data
`MemUsedMATLAB`	Total amount of memory used by the MATLAB process

View the value against the Total entry in the Virtual Memory section. It is shown as 2 GB in the table, which is the default on Windows XP systems. On UNIX systems, see the `ulimit` command to view and set user limits including virtual memory.

Disabling Java VM on Startup

On UNIX systems, you can increase the workspace size by approximately 400 MB if you start MATLAB without the Sun Java VM. To do this, use the command line option `-nojvm` to start MATLAB. This also increases the size of the largest contiguous block (and therefore the largest matrix) by about the same.

Using `-nojvm` comes with a penalty in that you will lose many features that rely on the Java software, including the entire development environment.

Starting MATLAB with the `-nodesktop` option does not save any substantial amount of memory.

Shutting down other applications and services (e.g., using `msconfig` on Windows systems) can help if total system memory is the limiting factor, but usually process limit (on 32-bit machines) is the main limiting factor.

Increasing System Swap Space

The total memory available to applications on your computer is comprised of physical memory (RAM), plus a *page file*, or *swap file*, on disk. The page or swap file can be very large, even on 32-bit systems (e.g., 16 TB (terabytes) on 32-bit Windows, 512 TB on 64-bit Windows). The operating system allocates the virtual memory of each process to either physical RAM or to this file, depending on its needs and those of other processes.

How you set the swap space for your computer depends on what operating system you are running on.

UNIX Systems

For more information about swap space, type `pstat -s` at the UNIX command prompt. For detailed information on changing swap space, ask your system administrator.

Linux Systems

You can change your swap space by using the `mkswap` and `swapon` commands. For more information on the above commands, type `man` followed by the command name at the Linux prompt.

Windows XP Systems

Follow the steps shown here:

1. Right-click the `My Computer` icon, and select **Properties**.

2. In the `System Properties` GUI, select the **Advanced** tab. In the section labeled **Performance**, click the **Settings** button.

3. In the `Performance Options` GUI, click the **Advanced** tab. In the section labeled **Virtual Memory**, click the **Change** button

4. In the `Virtual Memory` GUI, under **Paging file size for selected drive**, you can change the amount of virtual memory.

Using the 3GB Switch on Windows Systems

Microsoft Windows XP systems can allocate 3 GB (instead of the default 2 GB) to processes, if you set an appropriate switch in the `boot.ini` file of the system. The MathWorks recommends that you only do this with Windows XP SP2 systems or later. This gives an extra 1 GB of virtual memory to MATLAB, not contiguous with the rest of the memory. This enables you to store more data, but not larger arrays, as these are limited by contiguous space. This is mostly beneficial if you have enough RAM (e.g., 3 or 4 GB) to use it.

After setting the switch, confirm the new value of the virtual memory after restarting your computer and using the `memory` function.

```
[userview systemview] = memory;

systemview.VirtualAddressSpace
ans =
  Available: 1.6727e+009   % Virtual memory available to MATLAB.
      Total: 2.1474e+009   % Total virtual memory
```

For more documentation on this option, use the following URL:

`http://support.microsoft.com/support/kb/articles/Q291/9/88.ASP`

Similarly, on machines running Microsoft Windows Vista, you can achieve the same effect by using the command:

```
BCDEdit /set increaseuserva 3072
```

For more documentation on this option, use the following URL:

`http://msdn2.microsoft.com/en-us/library/aa906211.aspx`

Freeing Up System Resources on Windows Systems

There are no functions implemented to manipulate the way MATLAB handles Microsoft Windows system resources. Windows systems use these resources to track fonts, windows, and screen objects. Resources can be depleted by using multiple figure windows, multiple fonts, or several UI controls. One way to free up system resources is to close all inactive windows. Windows system icons still use resources.

12

Programming Tips

Introduction

This section is a categorized compilation of tips for the MATLAB programmer. Each item is relatively brief to help you browse through them and find information that is useful. Many of the tips include a reference to specific MATLAB documentation that gives you more complete coverage of the topic. You can find information on the following topics:

For suggestions on how to improve the performance of your MATLAB programs, and how to write programs that use memory more efficiently, see Improving Performance and Memory Usage

Command and Function Syntax

In this section...

Syntax Help

For help about the general syntax of MATLAB functions and commands, type

```
help syntax
```

Command and Function Syntaxes

You can enter MATLAB commands using either a *command* or *function* syntax. It is important to learn the restrictions and interpretation rules for both.

```
functionname arg1 arg2 arg3                % Command syntax
functionname('arg1','arg2','arg3')         % Function syntax
```

For more information: See Calling Functions in the MATLAB Programming Fundamentals documentation.

Command Line Continuation

You can continue most statements to one or more additional lines by terminating each incomplete line with an ellipsis (`...`). Breaking down a statement into a number of lines can sometimes result in a clearer programming style.

```
sprintf ('Example %d shows a command coded on %d lines.\n', ...
         exampleNumber, ...
```

```
            numberOfLines)
```

Note that you cannot continue an incomplete string to another line.

```
disp 'This statement attempts to continue a string ...
      to another line, resulting in an error.'
```

For more information: See Entering Long Statements in the MATLAB Desktop Tools and Development Environment documentation.

Completing Commands Using the Tab Key

You can save some typing when entering commands by entering only the first few letters of the command, variable, property, etc. followed by the **Tab** key. Typing the second line below (with **T** representing **Tab**) yields the expanded, full command shown in the third line:

```
f = figure;
set(f, 'papTuT,'cT)                     % Type this line.
set(f, 'paperunits','centimeters')    % This is what you get.
```

If there are too many matches for the string you are trying to complete, you will get no response from the first **Tab**. Press **Tab** again to see all possible choices:

```
set(f, 'paTT
PaperOrientation   PaperPositionMode  PaperType     Parent
PaperPosition      PaperSize          PaperUnits
```

For more information: See Tab Completion in the Command Window in the MATLAB Desktop Tools and Development Environment documentation

Recalling Commands

Use any of the following methods to simplify recalling previous commands to the screen:

- To recall an earlier command to the screen, press the up arrow key one or more times, until you see the command you want. If you want to modify the recalled command, you can edit its text before pressing **Enter** or **Return** to execute it.

- To recall a specific command by name without having to scroll through your earlier commands one by one, type the starting letters of the command, followed by the up arrow key.
- Open the Command History window (**View > Command History**) to see all previous commands. Double-click the command you want to execute.

For more information: See Recalling Previous Lines and Command History Window in the MATLAB Desktop Tools and Development Environment documentation.

Clearing Commands

If you have typed a command that you then decide not to execute, you can clear it from the Command Window by pressing the Escape (**Esc**) key.

Suppressing Output to the Screen

To suppress output to the screen, end statements with a semicolon. This can be particularly useful when generating large matrices.

```
A = magic(100);     % Create matrix A, but do not display it.
```

Help

In this section...

Using the Help Browser

Open the Help browser from the MATLAB Command Window using one of the following:

- Click the question mark symbol in the toolbar.
- Select **Help > Product Help** from the menu.
- Type the word `doc` at the command prompt.

Some of the features of the Help browser are listed below.

Feature	Description
Product Filter	Establish which products to find help on.
Contents	Look up topics in the Table of Contents.
Index	Look up help using the documentation Index.
Search	Search the documentation for one or more words.
Demos	See what demos are available; run selected demos.
Favorites	Save bookmarks for frequently used Help pages.

For more information: See Finding Information with the Help Browser in the MATLAB Desktop Tools and Development Environment documentation.

Help on Functions from the Help Browser

To find help on any function from the Help browser, do either of the following:

- Select the **Contents** tab of the Help browser, open the **Contents** entry labeled MATLAB, and find the two subentries shown below. Use one of these to look up the function you want help on.
  - Functions — Categorical List
  - Functions — Alphabetical List
- Type `doc functionname` at the command line.

Help on Functions from the Command Window

Several types of help on functions are available from the Command Window:

- To list all categories that you can request help on from the Command Window, just type

  ```
  help
  ```

- To see a list of functions for one of these categories, along with a brief description of each function, type `help` *`category`*. For example,

  ```
  help datafun
  ```

- To get help on a particular function, type `help functionname`. For example,

  ```
  help sortrows
  ```

Topical Help

In addition to the help on individual functions, you can get help on any of the following topics by typing `help topicname` at the command line.

Topic Name	Description
`arith`	Arithmetic operators

Topic Name	Description
relop	Relational and logical operators
punct	Special character operators
slash	Arithmetic division operators
paren	Parentheses, braces, and bracket operators
precedence	Operator precedence
datatypes	MATLAB classes, their associated functions, and operators that you can overload
lists	Comma separated lists
strings	Character strings
function_handle	Function handles and the @ operator
debug	Debugging functions
java	Using Sun Java from within the MATLAB software.
fileformats	A list of readable file formats
changeNotification	Microsoft Windows directory change notification

Paged Output

Before displaying a lengthy section of help text or code, put MATLAB into its paged output mode by typing `more on`. This breaks up any ensuing display into pages for easier viewing. Turn off paged output with `more off`.

Page through the displayed text using the space bar key. Or step through line by line using **Enter** or **Return**. Discontinue the display by pressing the **Q** key or **Ctrl+C**.

Writing Your Own Help

Start each program you write with a section of text providing help on how and when to use the function. If formatted properly, the MATLAB `help` function displays this text when you enter

```
help functionname
```

MATLAB considers the first group of consecutive lines immediately following the function definition line that begin with `%` to be the help section for the function. The first line without `%` as the left-most character ends the help.

For more information: See Help Text in the MATLAB Desktop Tools and Development Environment documentation.

Help for Subfunctions and Private Functions

You can write help for subfunctions using the same rules that apply to main functions. To display the help for the subfunction `mysubfun` in file `myfun.m`, type

```
help myfun>mysubfun
```

To display the help for a private function, precede the function name with `private/`. To get help on private function `myprivfun`, type

```
help private/myprivfun
```

Help for Methods and Overloaded Functions

You can write help text for object-oriented class methods implemented with M-files. Display help for the method by typing

```
help classname/methodname
```

where the file `methodname.m` resides in subdirectory `@classname`.

For example, if you write a `plot` method for a class named `polynom`, (where the `plot` method is defined in the file `@polynom/plot.m`), you can display this help by typing

```
help polynom/plot
```

You can get help on overloaded MATLAB functions in the same way. To display the help text for the `eq` function as implemented in `matlab/iofun/@serial`, type

```
help serial/eq
```

Development Environment

In this section...

Workspace Browser

The Workspace browser is a graphical interface to the variables stored in the MATLAB base and function workspaces. You can view, modify, save, load, and create graphics from workspace data using the browser. Select **View > Workspace** to open the browser.

To view function workspaces, you need to be in debug mode.

For more information: See MATLAB Workspace in the MATLAB Desktop Tools and Development Environment documentation.

Using the Find and Replace Utility

Find any word or phrase in a group of files using the Find and Replace utility. Click **View > Current Directory**, and then click the binoculars icon at the top of the **Current Directory** window.

When entering search text, you do not need to put quotes around a phrase. In fact, parts of words, like `win` for `windows`, will not be found if enclosed in quotes.

For more information: See Finding and Replacing Text in the Current File in the MATLAB Desktop Tools and Development Environment documentation.

Commenting Out a Block of Code

To comment out a block of text or code within the MATLAB editor,

1. Highlight the block of text you would like to comment out.
2. Holding the mouse over the highlighted text, select **Text > Comment** (or **Uncomment**, to do the reverse) from the toolbar. (You can also get these options by right-clicking the mouse.)

For more information: See Adding Comments in the MATLAB Desktop Tools and Development Environment documentation.

Creating M-Files from Command History

If there is part of your current MATLAB session that you would like to put into an M-file, this is easily done using the Command History window:

1. Open this window by selecting **View > Command History**.
2. Use **Shift+Click** or **Ctrl+Click** to select the lines you want to use. MATLAB highlights the selected lines.
3. Right-click once, and select **Create M-File** from the menu that appears. MATLAB creates a new Editor window displaying the selected code.

Editing M-Files in EMACS

If you use Emacs, you can download editing modes for editing M-files with GNU-Emacs or with early versions of Emacs from the MATLAB Central Web site:

```
http://www.mathworks.com/matlabcentral/
```

At this Web site, select **File Exchange**, and then **Utilities > Emacs**.

For more information: See General Preferences for the Editor/Debugger in the MATLAB Desktop Tools and Development Environment documentation.

M-File Functions

In this section...

M-File Structure

An M-File consists of the components shown here:

```
function [x, y] = myfun(a, b, c)   % Function definition line
% H1 line -- A one-line summary of the function's purpose.
% Help text -- One or more lines of help text that explain
%   how to use the function. This text is displayed when
%   the user types "help functionname".

% The Function body normally starts after the first blank line.
% Comments -- Description (for internal use) of what the
%   function does, what inputs are expected, what outputs
%   are generated. Typing "help functionname" does not display
%   this text.

x = prod(a, b);                    % Start of Function code
```

For more information: See Basic Parts of an M-File in the MATLAB Programming Fundamentals documentation.

Using Lowercase for Function Names

Function names appear in uppercase in MATLAB help text only to make the help easier to read. In practice, however, it is usually best to use lowercase when calling functions.

For M-file functions, case requirements depend on the case sensitivity of the operating system you are using. As a rule, naming and calling functions using lowercase generally makes your M-files more portable from one operating system to another.

Getting a Function's Name and Path

To obtain the name of an M-file that is currently being executed, use the following function in your M-file code.

```
mfilename
```

To include the path along with the M-file name, use

```
mfilename('fullpath')
```

For more information: See the `mfilename` function reference page.

What M-Files Does a Function Use?

For a simple display of all M-files referenced by a particular function, follow the steps below:

1. Type `clear functions` to clear all functions from memory (see Note below).

2. Execute the function you want to check. Note that the function arguments you choose to use in this step are important, since you can get different results when calling the same function with different arguments.

3. Type `inmem` to display all M-Files that were used when the function ran. If you want to see what MEX-files were used as well, specify an additional output, as shown here:

   ```
   [mfiles, mexfiles] = inmem
   ```

Note `clear functions` does not clear functions locked by `mlock`. If you have locked functions, (which you can check using `inmem`), unlock them with `munlock`, and then repeat step 1.

Dependent Functions, Built-Ins, Classes

For a much more detailed display of dependent function information, use the `depfun` function. In addition to M-files, `depfun` shows which built-ins and classes a particular function depends on.

Function Arguments

In this section...

Getting the Input and Output Arguments

Use `nargin` and `nargout` to determine the number of input and output arguments in a particular function call. Use `nargchk` and `nargoutchk` to verify that your function is called with the required number of input and output arguments.

```
function [x, y] = myplot(a, b, c, d)
disp(nargchk(2, 4, nargin))          % Allow 2 to 4 inputs
disp(nargoutchk(0, 2, nargout))      % Allow 0 to 2 outputs

x = plot(a, b);
if nargin == 4
   y = myfun(c, d);
end
```

Variable Numbers of Arguments

You can call functions with fewer input and output arguments than you have specified in the function definition, but not more. If you want to call a function with a variable number of arguments, use the `varargin` and `varargout` function parameters in the function definition.

This function returns the size vector and, optionally, individual dimensions:

```
function [s, varargout] = mysize(x)
nout = max(nargout, 1) - 1;
s = size(x);
for k = 1:nout
```

```
    varargout(k) = {s(k)};
end
```

Try calling it with

```
[s, rows, cols] = mysize(rand(4, 5))
```

String or Numeric Arguments

If you are passing only string arguments into a function, you can use MATLAB command syntax. All arguments entered in command syntax are interpreted as strings.

```
strcmp string1 string1
ans =
     1
```

When passing numeric arguments, it is best to use function syntax unless you want the number passed as a string. The right-hand example below passes the number 75 as the string, '75'.

```
isnumeric(75)                        isnumeric 75
ans =                                ans =
    1                                     0
```

For more information: See "Command vs. Function Syntax" on page 3-25 in the MATLAB Programming Fundamentals documentation.

Passing Arguments in a Structure

Instead of requiring an additional argument for every value you want to pass in a function call, you can package them in a MATLAB structure and pass the structure. Make each input you want to pass a separate field in the structure argument, using descriptive names for the fields.

Structures allow you to change the number, contents, or order of the arguments without having to modify the function. They can also be useful when you have a number of functions that need similar information.

Passing Arguments in a Cell Array

You can also group arguments into cell arrays. The disadvantage over structures is that you do not have field names to describe each variable. The advantage is that cell arrays are referenced by index, allowing you to loop through a cell array and access each argument passed in or out of the function.

Program Development

In this section...

Planning the Program

When planning how to write a program, take the problem you are trying to solve and break it down into a series of smaller, independent tasks. Implement each task as a separate function. Try to keep functions fairly short, each having a single purpose.

Using Pseudo-Code

You may find it helpful to write the initial draft of your program in a structured format using your own natural language. This *pseudo-code* is often easier to think through, review, and modify than using a formal programming language, yet it is easily translated into a programming language in the next stage of development.

Selecting the Right Data Structures

Look at what classes and data structures are available to you in MATLAB and determine which of those best fit your needs in storing and passing your data.

For more information: see Chapter 1, "Classes (Data Types)" in the Programming Fundamentals documentation.

General Coding Practices

A few suggested programming practices:

- Use descriptive function and variable names to make your code easier to understand.
- Order subfunctions alphabetically in an M-file to make them easier to find.
- Precede each subfunction with a block of help text describing what that subfunction does. This not only explains the subfunctions, but also helps to visually separate them.
- Do not extend lines of code beyond the 80th column. Otherwise, it will be hard to read when you print it out.
- Use full Handle Graphics property and value names. Abbreviated names are often allowed, but can make your code unreadable. They also could be incompatible in future releases of MATLAB.

Naming a Function Uniquely

To avoid choosing a name for a new function that might conflict with a name already in use, check for any occurrences of the name using this command:

```
which -all functionname
```

For more information: See the `which` function reference page.

The Importance of Comments

Be sure to document your programs well to make it easier for you or someone else to maintain them. Add comments generously, explaining each major section and any smaller segments of code that are not obvious. You can add a block of comments as shown here.

```
%---------------------------------------------
% This function computes the ... <and so on>
%---------------------------------------------
```

For more information: See Comments in the MATLAB Programming Fundamentals documentation.

Coding in Steps

Do not try to write the entire program all at once. Write a portion of it, and then test that piece out. When you have that part working the way you want, then write the next piece, and so on. It's much easier to find programming errors in a small piece of code than in a large program.

Making Modifications in Steps

When making modifications to a working program, do not make widespread changes all at one time. It's better to make a few small changes, test and debug, make a few more changes, and so on. Tracking down a difficult bug in the small section that you've changed is much easier than trying to find it in a huge block of new code.

Functions with One Calling Function

If you have a function that is called by only one other function, put it in the same M-file as the calling function, making it a subfunction.

For more information: See Subfunctions in the MATLAB Programming Fundamentals documentation.

Testing the Final Program

One suggested practice for testing a new program is to step through the program in the MATLAB debugger while keeping a record of each line that gets executed on a printed copy of the program. Use different combinations of inputs until you have observed that every line of code is executed at least once.

Debugging

In this section...

The MATLAB Debug Functions

For a brief description of the main debug functions in MATLAB, type

```
help debug
```

For more information: See Debugging Process and Features in the MATLAB Desktop Tools and Development Environment documentation.

More Debug Functions

Other functions you may find useful in debugging are listed below.

Function	Description
`echo`	Display function or script code as it executes.
`disp`	Display specified values or messages.
`sprintf`, `fprintf`	Display formatted data of different types.

Function	Description
`whos`	List variables in the workspace.
`size`	Show array dimensions.
`keyboard`	Interrupt program execution and allow input from keyboard.
`return`	Resume execution following a keyboard interruption.
`warning`	Display specified warning message.
`error`	Display specified error message.
`lasterr`	Return error message that was last issued.
`lasterror`	Return last error message and related information.
`lastwarn`	Return warning message that was last issued.

The MATLAB Graphical Debugger

Learn to use the MATLAB graphical debugger. You can view the function and its calling functions as you debug, set and clear breakpoints, single-step through the program, step into or over called functions, control visibility into all workspaces, and find and replace strings in your files.

Start out by opening the file you want to debug using **File > Open** or the `open` function. Use the debugging functions available on the toolbar and pull-down menus to set breakpoints, run or step through the program, and examine variables.

For more information: See Debugging Process and Features in the MATLAB Desktop Tools and Development Environment documentation.

A Quick Way to Examine Variables

To see the value of a variable from the Editor/Debugger window, hold the mouse cursor over the variable name for a second or two. You will see the value of the selected variable displayed.

Setting Breakpoints from the Command Line

You can set breakpoints with `dbstop` in any of the following ways:

- Break at a specific M-file line number.
- Break at the beginning of a specific subfunction.
- Break at the first executable line in an M-file.
- Break when a warning, or error, is generated.
- Break if any infinite or `NaN` values are encountered.

For more information: See Setting Breakpoints in the MATLAB Desktop Tools and Development Environment documentation.

Finding Line Numbers to Set Breakpoints

When debugging from the command line, a quick way to find line numbers for setting breakpoints is to use `dbtype`. The `dbtype` function displays all or part of an M-file, also numbering each line. To display delaunay`.m`, use

```
dbtype delaunay
```

To display only lines 35 through 41, use

```
dbtype delaunay 35:41
```

Stopping Execution on an Error or Warning

Use `dbstop if error` to stop program execution on any error and enter debug mode. Use `warning debug` to stop execution on any warning and enter debug mode.

For more information: See Backtrace and Verbose Modes in the MATLAB Programming Fundamentals documentation.

Locating an Error from the Error Message

Click on the underlined text in an error message, and MATLAB opens the M-file being executed in its editor and places the cursor at the point of error.

For more information: See Finding Errors, Debugging, and Correcting M-Files in the MATLAB Desktop Tools and Development Environment documentation.

Using Warnings to Help Debug

You can detect erroneous or unexpected behavior in your programs by inserting warning messages that MATLAB will display under the conditions you specify. See the section on Warning Control in the MATLAB Programming Fundamentals documentation to find out how to selectively enable warnings.

For more information: See the `warning` function reference page.

Making Code Execution Visible

An easy way to see the end result of a particular line of code is to edit the program and temporarily remove the terminating semicolon from that line. Then, run your program and the evaluation of that statement is displayed on the screen.

For more information: See Finding Errors, Debugging, and Correcting M-Files in the MATLAB Desktop Tools and Development Environment documentation.

Debugging Scripts

Scripts store their variables in a workspace that is shared with the caller of the script. So, when you debug a script from the command line, the script uses variables from the base workspace. To avoid errors caused by workspace sharing, type `clear all` before starting to debug your script to clear the base workspace.

Variables

In this section...

Rules for Variable Names

Although variable names can be of any length, MATLAB uses only the first N characters of the name, (where N is the number returned by the function `namelengthmax`), and ignores the rest. Hence, it is important to make each variable name unique in the first N characters to enable MATLAB to distinguish variables. Also note that variable names are case sensitive.

```
N = namelengthmax
N =
    63
```

For more information: See Naming Variables in the MATLAB Programming Fundamentals documentation.

Making Sure Variable Names Are Valid

Before using a new variable name, you can check to see if it is valid with the `isvarname` function. Note that `isvarname` does not consider names longer than `namelengthmax` characters to be valid.

For example, the following name cannot be used for a variable since it begins with a number.

```
isvarname 8thColumn
ans =
     0
```

For more information: See Naming Variables in the MATLAB Programming Fundamentals documentation.

Do Not Use Function Names for Variables

When naming a variable, make sure you are not using a name that is already used as a function name. If you do define a variable with a function name, you will not be able to call that function until you `clear` the variable from memory. (If it's a MATLAB built-in function, then you will still be able to call that function but you must do so using `builtin`.)

To test whether a proposed variable name is already used as a function name, use

```
which -all name
```

For more information: See Potential Conflict with Function Names in the MATLAB Programming Fundamentals documentation.

Checking for Reserved Keywords

MATLAB reserves certain keywords for its own use and does not allow you to override them. Attempts to use these words may result in any one of a number of error messages, some of which are shown here:

```
Error: Expected a variable, function, or constant, found "=".
Error: "End of Input" expected, "case" found.
Error: Missing operator, comma, or semicolon.
Error: "identifier" expected, "=" found.
```

Use the `iskeyword` function with no input arguments to list all reserved words.

Avoid Using i and j for Variables

MATLAB uses the characters `i` and `j` to represent imaginary units. Avoid using `i` and `j` for variable names if you intend to use them in complex arithmetic.

If you want to create a complex number without using `i` and `j`, you can use the `complex` function.

Avoid Overwriting Variables in Scripts

MATLAB scripts store their variables in a workspace that is shared with the caller of the script. When called from the command line, they share the base workspace. When called from a function, they share that function's workspace. If you run a script that alters a variable that already exists in the caller's workspace, that variable is overwritten by the script.

For more information: See M-File Scripts in the MATLAB Programming Fundamentals documentation.

Persistent Variables

To get the equivalent of a static variable in MATLAB, use `persistent`. When you declare a variable to be `persistent` within a function, its value is retained in memory between calls to that function. Unlike `global` variables, `persistent` variables are known only to the function in which they are declared.

For more information: See Persistent Variables in the MATLAB Programming Fundamentals documentation.

Protecting Persistent Variables

You can inadvertently clear `persistent` variables from memory by either modifying the function in which the variables are defined, or by clearing the function with one of the following commands:

```
clear all
clear functions
```

Locking the M-file in memory with `mlock` prevents any `persistent` variables defined in the file from being reinitialized.

Global Variables

Use global variables sparingly. The global workspace is shared by all of your functions and also by your interactive MATLAB session. The more global variables you use, the greater the chances of unintentionally reusing a variable name, thus leaving yourself open to having those variables change in value unexpectedly. This can be a difficult bug to track down.

For more information: See Global Variables in the MATLAB Programming Fundamentals documentation.

Strings

In this section...

Creating Strings with Concatenation

Strings are often created by concatenating smaller elements together (e.g., strings, values, etc.). Two common methods of concatenating are to use the MATLAB concatenation operator (`[]`) or the `sprintf` function. The second and third line below illustrate both of these methods. Both lines give the same result:

```
numChars = 28;
s = ['There are ' int2str(numChars) ' characters here']
s = sprintf('There are %d characters here', numChars)
```

For more information: See "Creating Character Arrays" on page 1-39 and Converting from Numeric to String in the MATLAB Programming Fundamentals documentation.

Comparing Methods of Concatenation

When building strings with concatenation, `sprintf` is often preferable to `[]` because

- It is easier to read, especially when forming complicated expressions
- It gives you more control over the output format
- It often executes more quickly

You can also concatenate using the `strcat` function, However, for simple concatenations, `sprintf` and `[]` are faster.

Store Arrays of Strings in a Cell Array

It is usually best to store an array of strings in a cell array instead of a character array, especially if the strings are of different lengths. Strings in a character array must be of equal length, which often requires padding the strings with blanks. This is not necessary when using a cell array of strings that has no such requirement.

The `cellRecord` below does not require padding the strings with spaces:

```
cellRecord = {'Allison Jones'; 'Development'; 'Phoenix'};
```

For more information: See Cell Arrays of Strings in the MATLAB Programming Fundamentals documentation.

Converting Between Strings and Cell Arrays

You can convert between standard character arrays and cell arrays of strings using the `cellstr` and `char` functions:

```
charRecord = ['Allison Jones'; 'Development  '; ...
              'Phoenix      '];
cellRecord = cellstr(charRecord);
```

Also, a number of the MATLAB string operations can be used with either character arrays, or cell arrays, or both:

```
cellRecord2 = {'Brian Lewis'; 'Development'; 'Albuquerque'};
strcmp(charRecord, cellRecord2)
ans =
     0
     1
     0
```

For more information: See Converting to a Cell Array of Strings and "String Comparisons" on page 1-59 in the MATLAB Programming Fundamentals documentation.

Search and Replace Using Regular Expressions

Using regular expressions in MATLAB offers a very versatile way of searching for and replacing characters or phrases within a string. See the help on these functions for more information.

Function	Description
`regexp`	Match regular expression.
`regexpi`	Match regular expression, ignoring case.
`regexprep`	Replace string using regular expression.

For more information: See "Regular Expressions" on page 2-59 in the MATLAB Programming Fundamentals documentation.

Evaluating Expressions

In this section...

Find Alternatives to Using eval

While the `eval` function can provide a convenient solution to certain programming challenges, it is best to limit its use. The main reason is that code that uses `eval` is often difficult to read and hard to debug. A second reason is that `eval` statements cannot always be translated into C or C++ code by the MATLAB Compiler.

If you are evaluating a function, it is more efficient to use `feval` than `eval`. The `feval` function is made specifically for this purpose and is optimized to provide better performance.

For more information: See MATLAB Technical Note 1103, "What Is the EVAL Function, When Should I Use It, and How Can I Avoid It?" at URL `http://www.mathworks.com/support/tech-notes/1100/1103.html`.

Assigning to a Series of Variables

One common pattern for creating variables is to use a variable name suffixed with a number (e.g., `phase1`, `phase2`, `phase3`, etc.). We recommend using a cell array to build this type of variable name series, as it makes code more readable and executes more quickly than some other methods. For example:

```
for k = 1:800
   phase{k} = expression;
end
```

Short-Circuit Logical Operators

MATLAB has logical AND and OR operators (&& and ||) that enable you to partially evaluate, or *short-circuit*, logical expressions. Short-circuit operators are useful when you want to evaluate a statement only when certain conditions are satisfied.

In this example, MATLAB does not execute the function myfun unless its M-file exists on the current path.

```
comp = (exist('myfun.m') == 2) && (myfun(x) >= y)
```

For more information: See "Short-Circuit Operators" on page 2-31 in the MATLAB Programming Fundamentals documentation.

Changing the Counter Variable within a for Loop

You cannot change the value of the loop counter variable (e.g., the variable k in the example below) in the body of a for loop. For example, this loop executes just 10 times, even though k is set back to 1 on each iteration.

```
for k = 1:10
   fprintf('Pass %d\n', k)
   k = 1;
end
```

Although MATLAB does allow you to use a variable of the same name as the loop counter within a loop, this is not a recommended practice.

MATLAB Path

In this section...

Precedence Rules

When MATLAB is given a name to interpret, it determines its usage by checking the name against each of the entities listed below, and in the order shown:

1. Variable
2. Subfunction
3. Private function
4. Class constructor
5. Overloaded method
6. M-file in the current directory
7. M-file on the path, or MATLAB built-in function

If you have two or more M-files on the path that have the same name, MATLAB selects the function that has its M-file in the directory closest to the beginning of the path string.

For more information: See "Function Precedence Order" on page 3-35 in the MATLAB Programming Fundamentals documentation.

File Precedence

If you refer to a file by its filename only (leaving out the file extension), and there is more than one file of this name in the directory, MATLAB selects the file to use according to the following precedence:

1. MEX-file
2. MDL-file (Simulink model)
3. P-Code file
4. M-file

For more information: See "Multiple Implementation Types" on page 3-37 in the MATLAB Programming Fundamentals documentation.

Adding a Directory to the Search Path

To add a directory to the search path, use either of the following:

- At the toolbar, select **File > Set Path**.
- At the command line, use the `addpath` function.

You can also add a directory and all of its subdirectories in one operation by either of these means. To do this from the command line, use `genpath` together with `addpath`. The online help for the `genpath` function shows how to do this.

This example adds `/control` and all of its subdirectories to the MATLAB path:

```
addpath(genpath('K:/toolbox/control'))
```

For more information: See Search Path in the MATLAB Desktop Tools and Development Environment documentation.

Handles to Functions Not on the Path

You cannot create function handles to functions that are not on the MATLAB path. But you can achieve essentially the same thing by creating the handles through a script file placed in the same off-path directory as the functions.

If you then run the script, using `run path/script`, you will have created the handles that you need.

For example,

1. Create a script in this off-path directory that constructs function handles and assigns them to variables. That script might look something like this:

```
File E:/testdir/createFhandles.m
   fhset = @setItems
   fhsort = @sortItems
   fhdel = @deleteItem
```

2. Run the script from your current directory to create the function handles:

```
run E:/testdir/createFhandles
```

3. You can now execute one of the functions by means of its handle.

```
fhset(item, value)
```

Making Toolbox File Changes Visible to MATLAB

Unlike functions in user-supplied directories, M-files (and MEX-files) in the *matlabroot*/toolbox directories are not time-stamp checked, so MATLAB does not automatically see changes to them. If you modify one of these files, and then rerun it, you may find that the behavior does not reflect the changes that you made. This is most likely because MATLAB is still using the previously loaded version of the file.

To force MATLAB to reload a function from disk, you need to explicitly clear the function from memory using `clear functionname`. Note that there are rare cases where `clear` will not have the desired effect, (for example, if the file is locked, or if it is a class constructor and objects of the given class exist in memory).

Similarly, MATLAB does not automatically detect the presence of new files in *matlabroot*/toolbox directories. If you add (or remove) files from these directories, use `rehash toolbox` to force MATLAB to see your changes. Note that if you use the MATLAB Editor to create files, these steps are unnecessary, as the Editor automatically informs MATLAB of such changes.

Making Nontoolbox File Changes Visible to MATLAB

For M-files outside of the toolbox directories, MATLAB sees the changes made to these files by comparing timestamps and reloads any file that has changed the next time you execute the corresponding function.

If MATLAB does not see the changes you make to one of these files, try clearing the old copy of the function from memory using `clear` *`functionname`*. You can verify that MATLAB has cleared the function using `inmem` to list all functions currently loaded into memory.

Change Notification on Windows

If MATLAB, running on Windows, is unable to see new files or changes you have made to an existing file, the problem may be related to operating system change notification handles.

Type the following for more information:

```
help changeNotification
help changeNotificationAdvanced
```

Program Control

In this section...

Using break, continue, and return

It's easy to confuse the `break`, `continue`, and `return` functions as they are similar in some ways. Make sure you use these functions appropriately.

Function	Where to Use It	Description
`break`	`for` or `while` loops	Exits the loop in which it appears. In nested loops, control passes to the next outer loop.
`continue`	`for` or `while` loops	Skips any remaining statements in the current loop. Control passes to next iteration of the same loop.
`return`	Anywhere	Immediately exits the function in which it appears. Control passes to the caller of the function.

Using switch Versus if

It is possible, but usually not advantageous, to implement `switch-case` statements using `if-elseif` instead. See pros and cons in the table.

switch-case Statements	if-elseif Statements
Easier to read.	Can be difficult to read.
Can compare strings of different lengths.	You need `strcmp` to compare strings of different lengths.
Test for equality only.	Test for equality or inequality.

MATLAB case Evaluates Strings

A useful difference between `switch-case` statements in MATLAB and C is that you can specify string values in MATLAB `case` statements, which you cannot do in C.

```
switch(method)
   case 'linear'
      disp('Method is linear')
   case 'cubic'
      disp('Method is cubic')
end
```

Multiple Conditions in a case Statement

You can test against more than one condition with `switch`. The first case below tests for either a `linear` or `bilinear` method by using a cell array in the case statement.

```
switch(method)
   case {'linear', 'bilinear'}
      disp('Method is linear or bilinear')
   case (<and so on>)
end
```

Implicit Break in switch-case

In C, if you do not end each `case` with a `break` statement, code execution falls through to the following `case`. In MATLAB, `case` statements do not fall

through; only one `case` may execute. Using `break` within a `case` statement is not only unnecessary, it is also invalid and generates a warning.

In this example, if `result` is `52`, only the first `disp` statement executes, even though the second is also a valid match:

```
switch(result)
   case 52
      disp('result is 52')
   case {52, 78}
      disp('result is 52 or 78')
end
```

Variable Scope in a switch

Since MATLAB executes only one `case` of any `switch` statement, variables defined within one `case` are not known in the other `cases` of that `switch` statement. The same holds true for `if-elseif` statements.

In these examples, you get an error when `choice` equals `2`, because `x` is undefined.

```
  -- SWITCH-CASE --
switch choice
   case 1
      x = -pi:0.01:pi;
   case 2
      plot(x, sin(x));
end
```

```
 -- IF-ELSEIF --

if choice == 1
   x = -pi:0.01:pi;
elseif choice == 2
   plot(x, sin(x));
end
```

Catching Errors with try-catch

When you have statements in your code that could possibly generate unwanted results, put those statements into a `try-catch` block that will catch any errors and handle them appropriately.

The example below shows a `try-catch` block within a function that multiplies two matrices. If a statement in the `try` segment of the block fails, control passes to the `catch` segment. In this case, the `catch` statements check the error message that was issued (returned by `lasterr`) and respond appropriately.

```
try
   X = A * B
catch
   errmsg = lasterr;
   if(strfind(errmsg, 'Inner matrix dimensions'))
      disp('** Wrong dimensions for matrix multiply')
end
```

For more information: See "The try-catch Statement" on page 8-17 in the MATLAB Programming Fundamentals documentation.

Nested try-catch Blocks

You can also nest `try-catch` blocks, as shown here. You can use this to attempt to recover from an error caught in the first `try` section:

```
try
   statement1                     % Try to execute statement1
catch
   try
      statement2                  % Attempt to recover from error
   catch
      disp 'Operation failed'     % Handle the error
   end
end
```

Forcing an Early Return from a Function

To force an early return from a function, place a `return` statement in the function at the point where you want to exit. For example,

```
if <done>
   return
end
```

Save and Load

In this section...

Saving Data from the Workspace

To save data from your workspace, you can do any of the following:

- Copy from the MATLAB Command Window and paste into a text file.
- Record part of your session in a `diary` file, and then edit the file in a text editor.
- Save to a binary or ASCII file using the `save` function.
- Save spreadsheet, scientific, image, or audio data with appropriate function.
- Save to a file using low-level file I/O functions (`fwrite`, `fprintf`, `...`).

For more information: See Saving the Current Workspace in the MATLAB Desktop Tools and Development Environment documentation, and "Using the diary Function to Export Data" on page 6-47 and "Using Low-Level File I/O Functions" on page 6-67 in the MATLAB Programming Fundamentals documentation.

Loading Data into the Workspace

Similarly, to load new or saved data into the workspace, you can do any of the following:

- Enter or paste data at the command line.
- Create a script file to initialize large matrices or data structures.

- Read a binary or ASCII file using `load`.
- Load spreadsheet, scientific, image, or audio data with appropriate function.
- Load from a file using low-level file I/O functions (`fread`, `fscanf`, `...`).

For more information: See Loading a Saved Workspace and Importing Data in the MATLAB Development Environment documentation, and "Using Low-Level File I/O Functions" on page 6-67 in the MATLAB Programming Fundamentals documentation.

Viewing Variables in a MAT-File

To see what variables are saved in a MAT-file, use `who` or `whos` as shown here (the `.mat` extension is not required). `who` returns a cell array and `whos` returns a structure array.

```
mydataVariables = who('-file', 'mydata.mat');
```

Appending to a MAT-File

To save additional variables to an existing MAT-file, use

```
save matfilename -append
```

Any variables you save that do not yet exist in the MAT-file are added to the file. Any variables you save that already exist in the MAT-file overwrite the old values.

Note Saving with the `-append` switch does not append additional elements to an array that is already saved in a MAT-file. See the example below.

In this example, the second `save` operation does not concatenate new elements to vector `A`, (making `A` equal to `[1 2 3 4 5 6 7 8]`) in the MAT-file. Instead, it replaces the 5 element vector, `A`, with a 3 element vector, also retaining all other variables that were stored on the first `save` operation.

```
A = [1 2 3 4 5];   B = 12.5;   C = rand(4);
save savefile;
```

```
A = [6 7 8];
save savefile A -append;
```

Save and Load on Startup or Quit

You can automatically save your variables at the end of each MATLAB session by creating a `finish.m` file to `save` the contents of your base workspace every time you quit MATLAB. Load these variables back into your workspace at the beginning of each session by creating a `startup.m` file that uses the `load` function to load variables from your MAT-file.

For more information: See the `startup` and `finish` function reference pages.

Saving to an ASCII File

When you save matrix data to an ASCII file using `save -ascii`, MATLAB combines the individual matrices into one collection of numbers. Variable names are not saved. If this is not acceptable for your application, use `fprintf` to store your data instead.

For more information: See "Exporting Delimited ASCII Data Files" on page 6-46.

Files and Filenames

In this section...

Naming M-files

M-file names must start with an alphabetic character, may contain any alphanumeric characters or underscores, and must be no longer than the maximum allowed M-file name length (returned by the function `namelengthmax`).

```
N = namelengthmax
N =
    63
```

Since variables must obey similar rules, you can use the `isvarname` function to check whether a filename (minus its `.m` file extension) is valid for an M-file.

```
isvarname mfilename
```

Naming Other Files

The names of other files that MATLAB interacts with (e.g., MAT, MEX, and MDL-files) follow the same rules as M-files, but may be of any length.

Depending on your operating system, you may be able to include certain nonalphanumeric characters in your filenames. Check your operating system manual for information on valid filename restrictions.

Passing Filenames as Arguments

In MATLAB commands, you can specify a filename argument using the MATLAB command or function syntax. For example, either of the following are acceptable. (The `.mat` file extension is optional for `save` and `load`).

```
load mydata.mat             % Command syntax
load('mydata.mat')          % Function syntax
```

If you assign the output to a variable, you must use the function syntax.

```
savedData = load('mydata.mat')
```

Passing Filenames to ASCII Files

ASCII files are specified as follows. Here, the file extension is required.

```
load mydata.dat -ascii              % Command syntax
load('mydata.dat','-ascii')         % Function syntax
```

Determining Filenames at Run-Time

There are several ways that your function code can work on specific files without you having to hardcode their filenames into the program. You can

- Pass the filename in as an argument

  ```
  function myfun(datafile)
  ```

- Prompt for the filename using the `input` function

  ```
  filename = input('Enter name of file:  ', 's');
  ```

- Browse for the file using the `uigetfile` function

  ```
  [filename, pathname] =
     uigetfile('*.mat', 'Select MAT-file');
  ```

For more information: See the `input` and `uigetfile` function reference pages.

Returning the Size of a File

Two ways to have your program determine the size of a file are shown here.

```
  -- METHOD #1 --                    -- METHOD #2 --
s = dir('myfile.dat');           fid = fopen('myfile.dat');
filesize = s.bytes               fseek(fid, 0, 'eof');
                                 filesize = ftell(fid)
                                 fclose(fid);
```

The `dir` function also returns the filename (`s.name`), last modification date (`s.date`), and whether or not it's a directory (`s.isdir`).

(The second method requires read access to the file.)

For more information: See the `fopen`, `fseek`, `ftell`, and `fclose` function reference pages.

Input/Output

In this section...

For more information and examples on importing and exporting data, see Technical Note 1602:

http://www.mathworks.com/support/tech-notes/1600/1602.html

File I/O Function Overview

For a good overview of MATLAB file I/O functions, use the online "Functions — Categorical List" reference. In the Help browser **Contents**, select **MATLAB > Functions — Categorical List**, and then click **File I/O**.

Common I/O Functions

The most commonly used, high-level, file I/O functions in MATLAB are `save` and `load`. For help on these, type `doc save` or `doc load`.

Functions for I/O to text files with delimited values are `textscan`, `dlmread`, `dlmwrite`. Functions for I/O to text files with comma-separated values are `csvread`, `csvwrite`.

For more information: See Text Files in the MATLAB "Functions — Categorical List" reference documentation.

Readable File Formats

Type `doc fileformats` to see a list of file formats that MATLAB can read, along with the associated MATLAB functions.

Using the Import Wizard

A quick method of importing text or binary data from a file (e.g., Excel files) is to use the MATLAB Import Wizard. Open the Import Wizard with the command, `uiimport filename` or by selecting **File > Import Data** at the Command Window.

Specify or browse for the file containing the data you want to import and you will see a preview of what the file contains. Select the data you want and click **Finish**.

For more information: See "Using the Import Wizard" on page 6-11 in the MATLAB Programming Fundamentals documentation.

Loading Mixed Format Data

To load data that is in mixed formats, use `textscan` instead of `load`. The `textscan` function lets you specify the format of each piece of data.

If the first line of file `mydata.dat` is

```
Sally    12.34 45
```

Read the first line of the file as a free format file using the % format:

```
fid = fopen('mydata.dat');
c = textscan(fid, '%s %f %d', 1);
fclose(fid);
```

returns

```
c =
    {1x1 cell}    [12.3400]    [45]
```

Reading Files with Different Formats

Attempting to read data from a file that was generated on a different platform may result in an error because the binary formats of the platforms may differ. Using the `fopen` function, you can specify a machine format when you open the file to avoid these errors.

Interactive Input into Your Program

Your program can accept interactive input from users during execution. Use the `input` function to prompt the user for input, and then read in a response. When executed, `input` causes the program to display your prompt, pause while a response is entered, and then resume when the **Enter** key is pressed.

Starting MATLAB

Getting MATLAB to Start Up Faster

Here are some things that you can do to make MATLAB start up faster.

- Make sure toolbox path caching is enabled.
- Make sure that the system on which MATLAB is running has enough RAM.
- Choose only the windows you need in the MATLAB desktop.
- Close the Help browser before exiting MATLAB. When you start your next session, MATLAB will not open the Help browser, and thus will start faster.
- If disconnected from the network, check the `LM_LICENSE_FILE` variable. See http://www.mathworks.com/support/solutions/data/1-17VEB.html for a more detailed explanation.

For more information: See Toolbox Path Caching in MATLAB in the MATLAB Desktop Tools and Development Environment documentation.

Operating System Compatibility

In this section...

Executing O/S Commands from MATLAB

To execute a command from your operating system prompt without having to exit MATLAB, precede the command with the MATLAB `!` operator.

On Windows, you can add an ampersand (`&`) to the end of the line to make the output appear in a separate window.

For more information: See Running External Programs in the MATLAB Desktop Tools and Development Environment documentation, and the `system` and `dos` function reference pages.

Searching Text with grep

`grep` is a powerful tool for performing text searches in files on UNIX systems. To `grep` from within MATLAB, precede the command with an exclamation point (`!grep`).

For example, to search for the word `warning`, ignoring case, in all M-files of the current directory, you would use

```
!grep -i 'warning' *.m
```

Constructing Paths and Filenames

Use the `fullfile` function to construct path names and filenames rather than entering them as strings into your programs. In this way, you always get the correct path specification, regardless of which operating system you are using at the time.

Finding the MATLAB Root Directory

The `matlabroot` function returns the location of the MATLAB installation on your system. Use `matlabroot` to create a path to MATLAB and toolbox directories that does not depend on a specific platform or MATLAB version.

The following example uses `matlabroot` with `fullfile` to return a platform-independent path to the `general` toolbox directory:

```
fullfile(matlabroot,'toolbox','matlab','general')
```

Temporary Directories and Filenames

If you need to locate the directory on your system that has been designated to hold temporary files, use the `tempdir` function. `tempdir` returns a string that specifies the path to this directory.

To create a new file in this directory, use the `tempname` function. `tempname` returns a string that specifies the path to the temporary file directory, plus a unique filename.

For example, to store some data in a temporary file, you might issue the following command first.

```
fid = fopen(tempname, 'w');
```

Demos

Demos Available with MATLAB

MATLAB comes with a wide array of visual demonstrations to help you see the extent of what you can do with the product. To start running any of the demos, simply type `demo` at the MATLAB command prompt. Demos cover the following major areas:

- MATLAB
- Toolboxes
- Simulink
- Blocksets
- Real-Time Workshop®
- Stateflow®

For more information: See Demos in the Help Browser in the MATLAB Desktop Tools and Development Environment documentation, and the `demo` function reference page.

For More Information

In this section...

Current CSSM

```
http://www.mathworks.com/matlabcentral/newsreader
```

Archived CSSM

```
http://mathforum.org/kb/forum.jspa?forumID=80
```

MATLAB Technical Support

```
http://www.mathworks.com/support/
```

Tech Notes

```
http://www.mathworks.com/support/tech-notes/list_all.html
```

MATLAB Central

```
http://www.mathworks.com/matlabcentral/
```

MATLAB Newsletters (Digest, News & Notes)

```
http://www.mathworks.com/company/newsletters/index.html
```

MATLAB Documentation

http://www.mathworks.com/access/helpdesk/help/helpdesk.html

MATLAB Index of Examples

http://www.mathworks.com/access/helpdesk/help/techdoc/demo_example.shtml

Index

Symbols and Numerics

A

D

F

M

N

W